MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN DATA SCIENCE

Classification of Linux Commands in SSH Session by Risk Levels

Thuy Ngan, Dào

Award date:
2020

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Classification of Linux Commands in SSH Session by Risk Levels

Dao Thuy Ngan

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Université de Namur
Faculté d'informatique
Année académique 2019–2020

Classification of Linux Commands in SSH Session by Risk Levels

Dao Thuy Ngan

Maître de stage : Jean-Noël COLIN

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Jean-Noël COLIN

Co-promoteur : Sereysethy Touch

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Honeypot is a decoy system with vulnerabilities introduced to trap hackers. Through many years of evolution, a generation of smart honeypots has been developed. The self-adaptive honeypot is a smart honeypot that is expected to respond appropriately to the attacker's request. In most existing self-adaptive honeypot systems, the commands sent from the attacker play a central role in the reasoning process of the honeypot. In this thesis, we focus on the construction of a machine learning workflow that aims at estimating the risk level of these commands. Experiments show that the proposed workflow achieves potential results.

Résumé

Honeypot est un système leurre avec des vulnérabilités introduites pour piéger les attaquants. Au cours de nombreuses années d'évolution, une génération de honeypots intelligents a été développée. Le self-adaptive honeypot est un honeypot intelligent qui devrait répondre de manière appropriée à la demande de l'attaquant. Dans la plupart des systèmes de self-adaptive honeypots existants, les commandes envoyées par l'attaquant jouent un rôle central dans le processus de raisonnement du honeypot. Ce mémoire se concentre sur la conception d'un workflow d'apprentissage automatique qui vise à estimer le niveau de risque de ces commandes d'entrée. Les expériences montrent que le workflow proposé produit des résultats potentiels.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Jean-Noël Colin, for providing valuable guidance and feedback throughout my internship, for his help to clear the subject of this thesis, for his patience and encouragement.

I would like to thank my co-supervisor, Mr. Touch , for his guidance in my first step in the project, for his welcoming in the lab, for his data to help me complete the experiments, for all our discussion and his review to help me complete this thesis.

Finally and most importantly, I would like to thank my warm family who always is there for me. This is the source of energy that helps me finish this work and will help me achieve more in the future.

Table of Contents

Chapter 1

Introduction

This project aims at defining a solution to estimate the risk level of the Linux commands that are sent from the attackers to the SSH honeypots. In the following section, the context and motivation will be explained. An preliminary analysis of the problem will be presented and the research questions will be identified in Sec. 1.2. An overview of the solution and the contributions of this thesis will finally be presented in Sec. 1.3

1.1 Context and Motivation

With the global spread of internet and the very fast development of technologies, more and more organisations digitize their business systems and connect them to the internet. This introduces new opportunities to our society and to the organisations themselves, but also new related risks. Security is always a great challenge that every organisation has to care about. While traditional methods such as IDS system or penetration testing can help secure the network, it should aware that vulnerabilities can still exist, and can be exploited at any time. Also, hackers never sit still. New methods and tactics to attack networks are developed day by day along with the development of the technologies. Thus, it is necessary to continuously study the attacks and find innovative ways of countering the threats. Honeypot is one of the techniques that serve these purposes.

Being studied since the late 80's, honeypots are decoy systems with intentionally introduced vulnerabilities that are deployed with the intent of attracting the hackers. In [3],

Cheswick told an interesting story about "a merry chase" of a cracker who fell into a trap system with fake services installed. In the literature, Cheswick's system is considered the first honeypot. Since then, the honeypots become more and more popular. Different honeypot systems have been developed for various purposes. However, the hackers don't sit still! They learn to detect the honeypots by studying the characteristics of different aspects in the target system: file system, available tools and running services, and the behaviour of the interaction such as the response time, the response content and format, etc. This is a long brain teaser game between the hackers and the administrators. To counter the hackers, a generation of smart honeypots, called self-adaptive honeypots, have been introduced.

As will be discussed in the chapter 2, self-adaptive honeypots are expected to respond appropriately to the attacker's input commands so that they can avoid being fingerprinted. At the same time, the underlying target system, if exists, must still be protected. In most existing self-adaptive honeypot systems, the commands sent from the attacker play a central role in determining the action to be taken, no matter what algorithm is applied in the learning process. Moreover, the input commands may have a direct impact on the underlying system in the high-interactive honeypots. Integrating a module to measure the risk of these commands may help these honeypots work smarter and protect their underlying environment when deal with dangerous inputs.

In this master thesis, the problem of automatically estimating the *risk level* of the Linux commands sent from the attackers to a honeypot is addressed. This work is conducted in the context of a master internship, which is a small support part of the ongoing doctoral project of Mr. Touch at the university of Namur. Inspired by the existing works about self-adaptive honeypots, Mr. Touch proposes a new conceptual architecture of Smart Honeypot. The core of this system is a Smart Proxy which contains a decision maker. The input commands and environmental information are used as the base to decide the action to be taken in return to the attacker's request. The internship is realized with the hope that the commands' risk level could be a useful supplementary information that helps augment the efficiency of the decision maker.

In the next section, a preliminary analysis of the problem and the identified research questions will be presented.

1.2 Preliminary analysis and research questions

The objective of this work is to construct a machine learning workflow which can estimate the risk level of a Linux command.

The problem of estimating the risk is in fact a prediction problem: given a Linux command, we need to predict its risk level. This is a supervised learning problem which required labeled data for training and testing purposes. During the internship, a set of honeypot data is provided. These data were recorded from two medium-interaction SSH honeypots which were configured and deployed by Mr. Touch . The sequences of command lines are then extracted from the data and grouped by SSH sessions to form a dataset. However, the available data are not yet labeled. A supplementary task is necessary: We need to estimate and assign the risk level for each command in the dataset.

By analyzing the problem and the available data, we identify four research questions:

- **RQ1:** The available data are in form of text sequences. However, machine learning algorithm requires numerical feature vectors. How to represent the text commands efficiently to use in machine learning algorithms?
- **RQ2:** Assigning the risk level to a command is a task that requires professional knowledge, including the understanding about the effect of the command and the ability to measure severity of that effect. This is a time-consuming task. Is there a way to label the data with a minimal human effort?
- **RQ3:** Which machine learning model will be appropriate for this task? Obviously, the most important criterion for choosing a model is its prediction performance. In addition, the prediction time should be as low as possible. Knowing that the model is for an interactive system: the attacker sends a command and waits for the response from the honeypot, the response time is very important.
- **RQ4:** In the security environment, bad predictions may be harmful. Which metric will be appropriate to measure different criteria of the system?

To tackle the problem, we firstly solve the simple case in which only 2 risk levels are defined: normal/benign (0) and risky/malicious (1). In this case, the problem becomes a binary classification problem. The general case with multiple risk levels is then addressed in form of a multi-class classification problem.

1.3 Principal Contributions

The main contribution of this work is the construction of a complete machine learning workflow to solve the problem of classifying the commands by risk levels. Through this workflow, all the research questions are addressed. This workflow consists of different phases: data collection and labeling (**RQ1**), data cleaning and pre-processing, data transformation (**RQ2**), model construction(**RQ3**), evaluation and interpretation of the predicted results (**RQ4**).

The original problem is considered in two cases. Firstly, we deal with the simple case where only two risk levels are defined: normal/benign (0) and risky/malicious (1). The problem is formulated as a binary classification problem. For this case, we focus on studying how to transform the text data into the numerical vectors. Our contribution here is to find out a good representation model for the Linux commands. By evaluating different representation models, we progressively improve the performance of the binary classification task. Our experiments show that the Doc2Vec model gives the best performance, we achieve a 97% accuracy on the test set of our data.

Secondly, we tackle the general case where multiple risk levels can be defined. This case is formulated as a multi-class classification problem. To label the dataset, we have to construct a model to automatically estimate the risk level. The contribution in this step is our solution of applying a novel *Labeling Model* to automatically estimate the risk level of the unlabeled commands. The basic idea is to take into account the weak supervision information such as the heuristic or the experts' knowledge about the Linux commands to describe how to estimate the risk level in a programmatic way. With the 5 estimated risk levels produced by the *Labeling model*, we achieved 99% accuracy on the test set.

This thesis is organized as follows:

Chap. 2 defines some background terms and summaries some related works.

Chap. 3 presents our proposed solution for the problem of classifying the commands in SSH session by multiple risk levels. Two workflows are introduced in Sec. 3.1 and Sec. 3.2, corresponding to the two case of the problem (the simple case with 2 risk levels, and the general case). In Sec. 3.3, the choice of evaluation metric will be discussed.

Chap. 4 presents the details of the workflow to classify the commands in the simple case (only 2 risk levels are identified: normal and malicious (risky)).

Chap. 5 presents the details of the workflow to classify the commands in the general case where multiple risk levels could be identified.

Finally, Chap. 6 discusses more about the non-functional requirements of our system such as the quick response time, the ability of maintenance and the interpretability of the prediction results. We also discuss the points to improve in the future work, particularly how to take advantage of the *Labeling Model* to infer more reliable label for the unlabeled dataset if we have a ground-truth of a small subset of labeled data annotated by human expert.

Chapter 2

Background and related works

In this chapter, an overview of the honeypots and self-adaptive honeypots will be presented in Sec. 2.1. Next, the terms “risk” and “risk level” which are the key words in the problem statement will be discussed in Sec. 2.2. Finally, some existing works related to the analysis of user commands will be summarized in Sec. 2.3.

2.1 Honeypots and Self-Adaptive Honeypots

2.1.1 Honeypots

The concept of honeypots has been introduced since the early 90’s but the term honeypot was only introduced for the first time in 2002 by Lance Spitzner [36]. Since then, several authors proposed different definitions for this term. In 2003, Fabien Pouget et al [25] offered a survey of the literature and introduced an intuitive definition: **A honeypot consists in an environment where vulnerabilities have been deliberately introduced in order to observe intrusions.**

The value of the honeypots relies in getting hacked. In industries, the honeypots are often deployed alongside the production systems. The intent of this strategy is to trick the hackers into hacking the decoy systems. The general purpose of honeypots is to make the attacker believe that he is interacting with a real machine. If this purpose is achieved,

this will allow the administrator to observe the behaviours of the attackers, then to guard against new attacks.

There are many ways to build and deploy a honeypot. To distinguish different types of honeypots, we can classify them by the level of interaction that they afford to attackers. The level of interaction gives us a scale with which to measure and compare honeypots. According to Spitzner [36], the more a honeypot can do and the more an attacker can do to a honeypot, the greater the information that can be derived from it. However, by the same token, the more an attacker can do to the honeypot, the more potential damage an attacker can do.

The honeypots are grouped into three interaction categories:

- **Low-interaction honeypots** refers to technologies that emulates certain predesignated services. They are passive (the honeypots are only listening) and allow only limited interaction for attacker or malware as there is no real operating system target that an attacker can operate on. This type of honeypot is evaluated as a safer and easy way to gather information about the frequently occurred attacks and their sources [35]. Likewise, it is relatively simple for an attacker to detect a low-interaction honeypot because of its functional limitation.

- **High-interaction honeypots** are the ones that give attackers access to a real operating system and its applications. They are extremely complicated to build and maintain. They are also at a very high level of risk. In returns, they can give a vast amount of information about attackers and their attacks because they are more attractive and they accept various types of attack. High-interaction honeypots have been mainly used to capture and analyze autonomous propagating malware such as worms, virus and botnets [19].

- **Medium-interaction honeypots** offer more interaction capabilities than low interaction honeypots but less functionalities than high-interaction honeypots. They are still the emulators, they do not provide any real underlying operating system. As a result, medium-interaction honeypots are safer than the high-interaction ones. In these honeypots, the daemons are designed to mimic the functionality of some real

applications when they interact with users. They give the attacker a better illusion of a real system and get more possibility to interact. This type of honeypot has many advantages, but developing such a system requires knowledge about the provided protocols and services. The more services a honeypot provides, the more complicated it is to deploy and maintain it.

The honeypots are popularly used in research and industries because of their many advantages. They can be used to detect unauthorized activities such as scanning. They can also be configured and deployed to capture the latest worm for analysis. The honeypots collect data that records only malicious activities. This is a data source of high value because there is very little noise. These data can be used for analyzing attacks, profiling attackers or training a smart security system.

2.1.2 Self-adaptive Honeypots

With the application of machine learning techniques, honeypots are becoming smarter. Self-adaptive honeypots refer to the smart honeypots that can learn to make decision of actions to be taken while interacting with attackers.

The first self-adaptive honeypot was introduced in 2011 by Wagener et. al. [37]. That is a high-interaction honeypot which exposes a SSH service. The system leverages the game-theoretic concepts and a variant of reinforcement learning method to learn to interact with the hackers. It can decide to execute the received command or perform another predefined action with a probability detected by reinforcement learning. The experimental results show that behavioral strategies are dependent on contextual parameters and can serve as advanced building blocks for intelligent honeypots.

Inspired by the system of Wagener, some variants of the self-adaptive honeypot with adaptations and improvements have been presented. Pauna et. al. [23] proposed a case-adaptive honeypot, called CASSH, based on the existing medium-interaction honeypot Kippo. The honeypot was designed using the Beliefs-Desires-Intentions agent model with the learning capabilities of Case Base Reasoning technique, i.e. the actions are planned using the accumulated experience learned from the similar tasks. In the next step, Pauna

and Bica created a new changing behavior honeypot system called RASSH [20]. In this system, the machine learning SARSA algorithm is used in the reinforcement learning module to decide what action to be taken in responding to the attacker's command. According to the authors, this system overlaps some of the disadvantages in the existing systems. Recently, the same author proposed an improvement with the use of deep Q-learning (DQN) algorithm to fully automate the decision process for a self-adaptive SSH honeypot [22] and an IoT honeypot [21].

Besides the works of Pauna, in a study by Luo et. al. [15], an "intelligent-interaction" honeypot named IoTCandyJar was introduced for IoT devices. This system learns the behaviours of IoT devices using a set of potential responses for the captured requests collected from different public available IoT devices. To pass attacker's checks, multiple heuristics and reinforcement learning mechanism are applied to help the system learn the best responses which has a high probability to be the one that the attackers wait for. Their result shows that the honeypot is improved in term of session length and number of captured attacks.

In another work, Dowling et. al. proposed another improvement of the self-adaptive honeypot with a new state-action space formalism that aims at interact with the automated malware [6] [5].

In most self-adaptive honeypot systems, including the target system of this work, the commands sent by attackers play an important role. They are the primary mean to describe the interaction situation, which influences directly the decision on which action to take. Analyzing these commands and deriving useful information such as their risk level can then be helpful to improve the performance and the effectiveness of those systems, especially the SSH honeypots. In Sec. 2.3, some studies on the analysis of user commands in SSH compromises will be discussed.

2.2 Risk Level Definition

According to the EBIOS Risk Manager method guide [9], the term "risk" and "risk level" are defined as follows:

- Risk: Possibility of a feared event occurring and that its effects affect the missions of the studied object.
- Risk Level: Measurement of the extent of the risk, expressed by combining the severity and the likelihood.

The term "feared event" which is used in the definition of risk is also defined in the EBIOS guide:

- A feared event is associated with a business asset and harms a security need or criterion of the business asset (examples: unavailability of a service, illegitimate modification of a high temperature threshold of an industrial process, disclosure of classified data, modification of a database). The feared events to be exploited are those of the strategic scenarios and relate to the impact of an attack on a business asset. Each feared event is assessed according to the level of severity of the consequences, using metrics.

Let identify the feared event in our context. This work concentrates on evaluating the risk level of the Linux commands sent from an attacker to a honeypot after he successfully logged in to the system with a SSH username and password. Then the feared event is the action "send a command to the system" of the attacker. At the moment when the system evaluates the command, it's obvious that the command was sent successfully (so that the honeypot received and will process it). Then, the possibility of the feared event occurring is 1. By consequence, the estimation of the risk level of a command becomes the evaluation of its severity.

It is not easy to estimate the risk level of each Linux command. Firstly, this task requires expert knowledge and experience to correctly evaluate how a command can make change to a system. Secondly, the severity of executing a command depends heavily on the state of its environment. For example, someone types a command `rm` to delete a file in the system. In the terminal (or the bash history), only the command name and file name present. But the result of the action depends on many environmental factors. Does the files exist in the current directory? Does the current user have the write permission on it? If the file doesn't exist or the user doesn't have the write permission, then the command

"rm" doesn't have any impact on the system. By contrast, that command can make a catastrophe if the target file is important.

In this work, the environmental factors are not yet taken into account (but the command parameters are still considered). We consider only the impact that a command can cause to the system when all necessary conditions are satisfied for it to be successfully executed. With this assumption, the command "rm" is always dangerous because an attacker can always delete any file that he wants.

2.3 Related works on the analysis of SSH compromises

Since their appearance, the honeypots have been providing a high quality source of malicious data for analysis. Various works have been published, revealing the attacking patterns and the behaviour of the attackers.

Despite many researches on attack data, few works exploit the sequences of commands sent from the attackers. In [27], Ramsbrock et. al. presented a method to profile the behaviour of the attackers by exploiting the snapshots of Linux commands captured from their high-interaction honeypots. The authors identified seven groups of commands representing the typical actions that an attacker can take: check software configuration, install a program, download a file, run a rogue program, change password, check hardware configuration and change the system configuration. The attacker profile is then represented in form of a state machine where the actions are represented by the states and their orders are represented by the transitions.

In [7], Dumont et. al. leverages the malicious Linux commands recorded from their SSH honeypots to learn a system that can detect the malicious remote Shell sessions. The collected honeypot session logs are used together with the content of the `.bash_history` files crawled from github.com to form a data set for training and testing purpose. The n-grams are then extracted from the sequences of 1-4 consecutive commands and used as features to train a k-nearest-neighbors classifier. Even if the proposed pipeline is quite simple, the experimental results are impressive: the classifier reaches a true positive rate of 99.4% and a true negative rate of 99.7% with sequences of four shell commands.

The solution proposed by Dumont et. al. [7] is for detecting the malicious SSH **sessions**. In the problem of classifying **commands** by level of risk, this model can be useful when the number of risk levels is reduced to 2, i.e a command could be evaluated to be one of the two case: safe (benign) or risky (malicious). The approach of Dumont is then re-implemented as a part of the proposed solution and plays the role of a baseline model. Next, this baseline will progressively be improved by introducing different representation models for text data. Later, an original solution to the full problem of classifying the commands by **multiple risk levels** will be proposed without requiring hand-labeled data. The method reasoning and the implementation are presented in the chapter 3, 4 and 5 of the thesis.

Summary Chapter *Background and related works*

In this chapter, the definitions of honeypot and self-adaptive honeypot are presented.

The honeypots can be classified by their level of interaction, there are 3 categories: low-interaction honeypot, high-interaction honeypot and medium interaction honeypot.

The definitions of the terms risk and risk level are also presented. The risk level of a command sent to the honeypot is its degree of severity.

The estimation of the risk level of a command is complicated because of the underlying environment state. In this work, we consider only the impact that a command can cause to the system when all necessary conditions are satisfied for it to be successfully executed.

This chapter also summaries some existing works related to the analysis of user commands. Some of them give us inspiration to elaborate the solution that will be presented in Chapter 3.

Chapter 3

Solution Elaboration

This chapter presents our proposed solution for the problem of classifying the commands in SSH session by multiple risk levels. We consider two cases in this thesis. In the simple case where there are only two risk levels: normal (level 0) and risky (level 1), our problem is a binary classification problem. By discriminating these two class, we indeed solve the problem of detection of malicious commands. In the general case, we need to classify a command by different risk levels, that is a multi-class classification problem. Another approach to this problem is to use the syntax analyzer to analyze the Linux command. A traditional rule-based method can then be used to filter the processed commands through a set of specific rules. The disadvantage of this approach will be discussed in Chap. 5 when we have multiple risk levels to estimate for each command. The biggest advantages of Machine Learning approach is that (i) we do not have to maintain a complicated set of rules (ii) we can define clearly the performance metrics to evaluate the models, (iii) we can generalize (make prediction) with unseen data.

Our work focuses on building a **complete workflow** to accomplish the tasks defined above. Applying the machine learning methods for classification is only on step in the whole workflow. The other important parts are: data collection, data labeling, data cleaning and pre-processing, data transformation, evaluating of the predictive model and interpreting the results. The complete workflows for the two cases are presented in details in Sec. 3.1 and Sec. 3.2. The most important parts of our work are in (i) the representation

learning for the text data and (ii) the procedure of automatically building the *Label Model* for labeling the dataset.

3.1 Workflow for the Detection of Malicious Commands

We first consider the problem of classifying the commands by risk levels in a special case where there are only two levels: normal (benign, label 0) and risky (malicious, label 1). It now becomes a binary classification problem of identifying if a command is risky/malicious (class positive) or normal/benign (class negative). It can also be considered as a problem of detecting the malicious commands. [1] The proposed workflow for solving this problem is illustrated in Fig. 3.1.

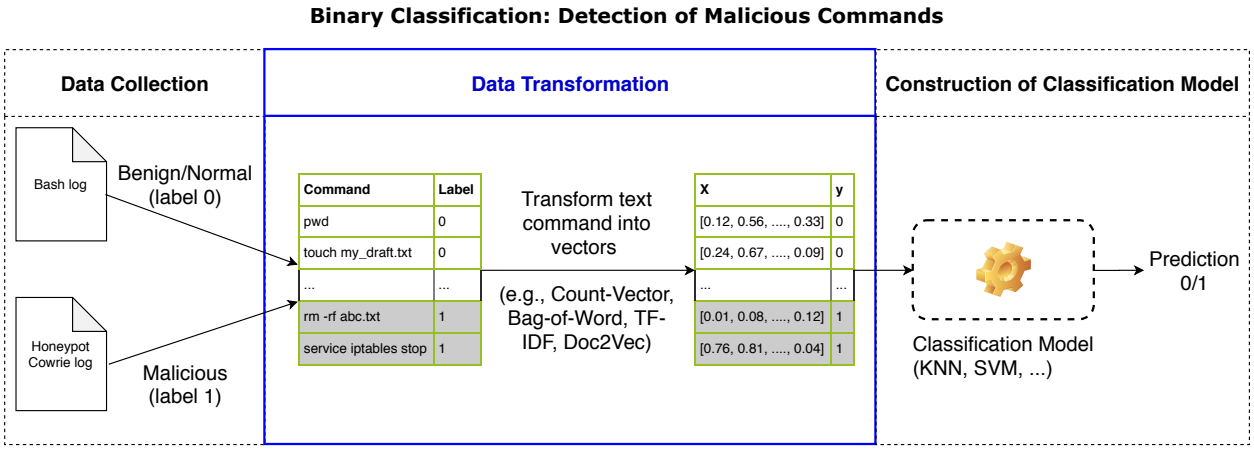


Figure 3.1: Workflow of binary classification problem for detecting malicious commands. The most important part is how to transform the text data into numerical data (vector) to use in the Machine Learning algorithms.

This workflow is inspired by the work of Dumont et. al [7]. In brief, we first need to collect example commands for both classes, then transform the text commands into the numerical vectors. Different representation models to transform the text commands are reviewed including the traditional Bag-of-Word model [11], the TF-IDF model [12] and

[1] The name negative, positive class come from the analogy in medical testing. If a patient is infected by a disease, we say the patient is *positive* to the disease X. In the binary classification, the positive class is assigned label 1 and the negative class is assigned level 0 (or -1, but in our work, we use label 0, that can be interpreted as zero risk level, i.e. no risk.

the modern Doc2Vec model [13]. These models are explained in great details in Chap. 4. Combining these representation models with different classification models gives us different pipelines for the malicious commands detection task. These pipelines are evaluated with the guideline presented in Sec. 3.3.

3.2 Workflow for the Classification of Commands by Multiple Risk Levels

The proposed workflow for the problem of classification of commands by multiple risk levels is illustrated in Fig. 3.2.

In this problem, we will work uniquely on the commands captured on the two honeypots deployed by Mr. Touch in his doctoral project. The classification task is a supervised learning problem, so the labeled data are required for training and testing purpose. In our case, the labeled data (the risk level associated with each command) are not yet available. In order to build the dataset for a supervised learning model, we first need to label the data, i.e assign the risk level to each command in the dataset. Hand-labeling a large amount of text commands is a time-consuming task that requires the expert knowledge. We propose to use the *Labeling Model* [28], a novel approach to build the data programmatically and automatically with weak supervision.

As illustrated in Fig. 3.2, in the *Data Collection* phase, we firstly do the exploratory data analysis (by using an unsupervised machine learning method such as topic modeling) to gain some insights about the dataset. We also benefit from some prior knowledge about the Linux commands (of an expert or of the Linux users) to describe the rules of how to estimate - no need to very accurate - the risk level of a particular command. We then write an ensemble of *labeling functions* (in Python) to encode these knowledge and heuristics and construct a *Labeling Model*. The result model will automatically assign the label for each unlabeled command in the dataset.

In the next phase, the text data are transformed into vector of features. This phase follows the similar workflow as in the binary classification problem. The result training

vectors will then be fit to a multi-class classification model. Since the *labeling functions* are flexible and very easy to maintain, based on the performance of the classification, we can turn back to update the heuristics and improve the *Labeling Model*.

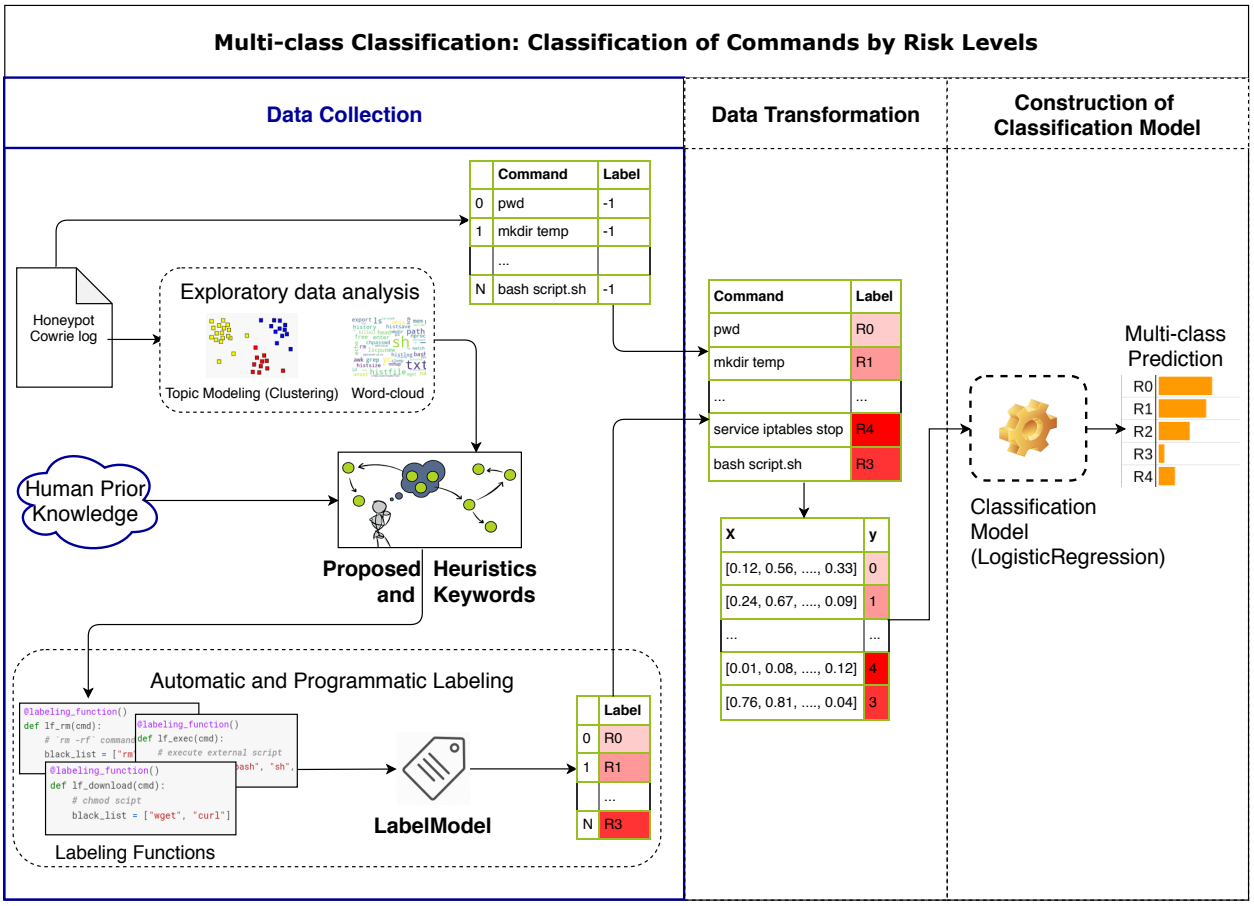


Figure 3.2: Workflow of multi-class classification problem for classifying the commands by risk levels. The most important part is the procedure to estimate risk level for the unlabeled commands using the novel *Labeling Model*.

3.3 Evaluation of Classification Models

Typically in a classification problem, the performance of a classifier can be measured by the *accuracy* metric. The accuracy is calculated in function of the number of data points in the test set that are correctly classified. This is the standard metric for any kind of classification model. However, for the binary classification model for sensitive applications such

as fraud transaction detection, cancer diagnosis or our malicious command detection, we need another metric. Let think of the following examples: a fraud transaction is predicted as legit and is processed, a cancer patient is diagnosed as just having a benign tumor and goes home without any special treatment, and a dangerous malicious command is classified as normal and allowed to be executed in the real system. These are some examples of bad prediction in the binary classification problem, where the positive value (fraud, cancer, malicious) are predicted as negative (legit, benign, normal). In these cases, this kind of inaccurate prediction can cause harmful consequences. As the accuracy of a machine learning model never reaches 100%, for the sensitive applications, another metric called confusion matrix is useful to estimate the inaccurate prediction part in more detail.

Table 3.3 explains the content of a confusion matrix. The definition and representation of confusion matrix are sometimes confused, particularly for the binary classification. Here we use the notation in the Machine Learning textbook of Murphy [18].

		Predicted Label	
		Negative (0)	**Positive (1)**
True Label	**Negative (0)**	True Negative	False Positive
	Positive (1)	False Negative	True Positive

Figure 3.3: Definition of confusion matrix for the classification problem.

The following metrics are reported in the confusion matrix:

- ***True Negative***: Real negative is predicted as negative.
- ***True Positive***: Real positive is predicted as positive.
- ***False Positive***: Real negative is predicted as positive.
- ***False Negative***: Real positive is predicted as negative.

The *True Negative* and *True Positive* reflect the accuracy of the prediction and are expected to be high. The *False Positive* error is also called *false alarm* error and often less critical than

the *False Negative* error in our specific problem. It should note that, the criterion of these metrics depends on the problem. Let us consider another problem of spam detection. The spam emails are labeled positive (class 1) and the non-spam emails are labeled negative (class 0). The goal of a spam detector is to prevent the spam emails from appearing in the primary mail box, instead put them in the spam folder. In practice, if some spam emails are classified as normal (*False Negative* error), we will be a little annoyed seeing the spam emails in the mail box, but that is not a big problem. On the contrary, if an important working emails is classified as spam (*False Positive* error), we can not see them in the mail box and we may miss some important information. In this case, the *False Positive* is more critical than the *False Negative* and should be reduced as much as possible.

Back to our binary classification model, the target is to increase the accuracy while maintaining a low *False Negative* error. There are also some other suggested targets for the binary classification such as increasing the *recall* of the positive class and increasing the *precision* of the negative class. However these targets are not clear and do not show the natural of the problem. Choosing the appropriate metrics for the problem is an important step, and defining the evaluation metrics beforehand helps us focus on the target. [2] (See more details in Chap. 4)

The confusion matrix and the accuracy metric are also used for the problem of classification by multiple risk levels. In this task, the labeled data are not available. We thus construct a *Labeling Model* to automatically assign the labels for our dataset. The target in this task is to increase the accuracy and reduce both *False Negative* and *False Positive* errors. (See more details in Chap. 5).

[2] We do not simply apply the machine learning model on the available data and report the results. Instead, we define the target for our system first, and then collect data, train the model towards this target.

Summary Chapter *Solution Elaboration*

We are constructing a workflows for a system, that can (i) automatically detect if an input command is benign/normal or malicious and (ii) tell how risky an input command is. We propose to formulate these tasks as classification problems:

- A *Binary Classification* problem to classify an input command as normal (class 0) or malicious (class 1).
- A *Multi-class Classification* problem to assign a particular risk level for each input command.

We build different Machine Learning pipelines to solve these two problems. In order to make a Machine Learning solution work, we need labeled data, we need to train a model to learn from data so we can output prediction on the unseen data.

- We first collect the data, label them, pre-process and transform them into numerical vectors.
- We train different classification models, explain how to evaluate these model and draw attention to the importance of reducing the *False Negative* error.

Our contribution in this work is 3-fold:

- We construct a complete machine learning workflow to solve the problem of classifying the commands by multiple risk levels, which is the objective of this work.
- We evaluate different ways to represent the input text commands and find that the Doc2Vec model works best. We achieve around 97% accuracy for the binary classification.
- We propose to use *Labeling Model*, an automatic and programmatic way to label a large dataset that provide reliable labels. That help us achieve around 99% accuracy for the multi-class classification with five defined risk levels.

Chapter 4

Detection of Malicious Commands in SSH Session

The first problem we are dealing with is to detect the malicious commands. Indeed, this is the simple case in our main problem of classifying the commands by risk level where there are only two risk levels: 0 (normal, benign) and 1 (risky, malicious). With this formulation, we make an assumption saying that, the commands in the working sessions of normal users are considered benign and the commands in the SSH sessions of the attackers captured on the honeypots is considered malicious.

If we have enough good examples for the benign and the malicious commands, we can train a binary classification model to discriminate these two classes. As discussed in Sec. 2.3, the work of Dumont et. al. [7] fits quite well with this problem. We therefore re-implement the model of Dumont as part of our solution and use it as our baseline model. First, the commands from different sources are collected to form the dataset (Sec. 4.1). Second, the representation models are built to transform the text data into feature vectors. In Sec. 4.2 we review the simple Bag-of-Word model followed by some more robust and widely used models (N-command, TF-IDF and Doc2Vec). These representation models are experimented with different classification models in Sec. 4.3. We analyze in depth the performance of the proposed workflow based on the accuracy and the *False Negative* error.

4.1 Data Collection and Pre-processing

Since the classification models require labeled data, we have to collect sufficient example for both classes benign and malicious. In our experiments, two sources of data are used.

- Benign example commands are collected from public bash log files in Github. They are the history of input commands of normal users in their working sessions. We crawl 660 long `.bash_history` files containing $210,402$ commands. The label 0 (negative class) are assigned to these commands to indicate that they are benign (coming from normal users).
- Malicious example commands are collected from the two honeypots by Mr. Touch . These data include $694,643$ commands of $76,778$ short sessions. All these commands are assigned the label 1 (positive class indicating malicious/risky commands).

There are $905,045$ commands in total, 70% of them is used to train the classification and the remaining 30% ($271,514$ commands) is reserved for the test set. Following the workflow proposed in Fig. 3.1, we focus on building representation models to transform the text commands into vectors and building classification models for detecting malicious commands. The representation model and classification model are trained and tuned on the training set. In the prediction step, the test data is first passed through the representation model to obtain the feature vectors. These vectors are then fed to the classification model to get the final prediction result. [1]

The data representation models presented in this chapter are also used in the workflow of the classification of the commands by multiple risk-levels (Chap. 5). Before building the representation models, a simple pre-processing step is applied to the input text commands in order to remove the unique strings like random generated file names, long file paths, email addresses, IP addresses and URLs.

[1] We do not use a separate validation set since training a complex model like Doc2Vec takes time. We tune the hyper-parameters of the models mostly by grid search with several common values. We simply compare the accuracy on the training and on the test set to avoid overfitting.

4.2 Representation Learning for text data

The text command is comprehensive to human, however, in order to analyze them using Statistical Machine Learning techniques, we have to perform mathematical operations on them. Thus the text data must be transformed into numerical vectors (which is called the *feature vectors*). Several basic techniques in Natural Language Processing (NLP) are applied to process the textual commands.

4.2.1 Basic Concepts

Our dataset is a collection of Linux commands. Each command is called a ***document***. A document is in fact a sequence of text, it can be as long as a book/an article or as short as a message/a tweet. In our context, basically, we treat each command as a document. We can also treat one command and several previous commands as a document. A document can be ***tagged***, that means it can be accompanied by a list of tags to indicate different labels/properties assigned to it. For the problem of classification the command by risk level, each command is tagged by it corresponding estimated risk level.

When processing a document, it can be ***tokenized***, i.e., separated into ***tokens*** by some rules (such as breaking at the end of each sentence or breaking by the white space). In our application, a document is one or several commands glued together, the tokenizer simply breaks a document into ***words*** or ***terms***. For example, tokenzing the command `pip install --upgrade gensim` gives a list of tokens (or ***unigram***)
`['pip', 'install', '--upgrade', 'gensim']`.
It can be useful if we care about not only individual word but also a group of consecutive words. For example, for the command given above, if we would like to extract all pairs of two consecutive words (called ***bigram*** or ***2-gram***), we have
`['pip_install', 'install_--upgrade', '--upgrade_gensim']`.
Similarly, if we extract a tuple of three words (called ***trigram*** or ***3-gram***), we have
`['pip_install_--upgrade', 'install_--upgrade_gensim']`.

The whole dataset is called the ***corpus***. The corpus can be understand as a standard way to organize the data, which is used by the ***model*** such as a topic model (which looks

for the hidden topics/themes in the corpus). In general, the most common task in NLP is to infer the latent (hidden) structures in the corpus. These structures can be the groups of documents with close semantic meaning or the classes of documents with the same tag. In order to do that, the corpus must be manipulated mathematically and numerically. By characterizing a document, we can represent each document as a vector of its features. Different ways to characterize the document give us different representation models.

4.2.2 Document Representation Model

The most fundamental and most important document representation model is the Bag-of-Words (BoW) model. In the early 50s, Zellig Harris, an influence linguist found that, "*each language can be described in terms of a distributional structure, i.e. in terms of the occurrence of parts relative to other parts*" [11]. The BoW model represents each document by the occurrence (frequency) of each word.[2] First, a dictionary of vocabulary is constructed from all the words in the corpus. The dictionary contains all the unique tokens with their frequency in the corpus. The dictionary can be made richer by using n-gram (normally 2-gram or 3-gram) tokens. (In that way, we can have a dictionary that contains at the same time unigram, 2-gram and/or 3-gram and so on). As the dictionary can be very large, we should consider to limit its size D, set several rules to ignore the stop words, to filter out the strange words that appears very few times, etc. The BoW model is exactly this dictionary (of size D) with the optimal data structure to store and retrieve data efficiently. A document is encoded by a vector of size D (called a ***count-vector***). The indices of element in the vector correspond to the words stored in the dictionary (that depends on the data structure), and the values are the number of occurrence of the corresponding word.

Despite its simplicity, the BoW model is still useful in many situations, particularly in our application, due to two reasons. First, the Linux commands (a document in BoW model) are often non-ambiguous. A command has an unique name and does specific,

[2]It should clearly note that, the occurrence is the count of how many times a word appears in the documents, it is an integer number. The frequency is the count normalized, and is float number. But in a general BoW model, we can use two terms interchangeably. We will specify the usage of frequency in other (TF-IDF) model.

deterministic task(s). It is distinguishable by its name (and/or some specific parameters). That makes a document in a BoW model distinguishable by specific word(s) in the dictionary. Second, the set of Linux commands is limited and very small in comparison to the set of vocabulary in human language, thus it makes the construction of BoW very fast and efficient. However, the largest disadvantage of BoW model is that the order of the words in a document is completely ignored. We can overcome this shortcoming by using n-gram tokens ($n > 1$) to specify the order of words (which is usually called the ***spatial information***).

This idea is also applied in computer vision and usually called *Bag of Visual Features* [38] model. This approach bases on hand-crafted visual features like *Scale Invariant Feature Transform (SIFT)* [14] feature, which is the vector of mathematical descriptions of local *keypoints* in an image. Instead of representing the image by a matrix of pixels, it can be represented by a histogram of visual features, which brings more semantic meaning for the vision tasks.

4.2.3 More Robust Representation Models

The basic BoW model represents the document by ***count-vector***. The first improvement discussed above is to use n-gram to specify the order of the words. For example, we can create a BoW model with all unigram, bigram and trigram for each document (denoted as $\{1, 2, 3\}$-gram). The second improvement is to represent the command in its context. We use the notation 1**-command** to denote that only one current command is processed at a time. N**-command** notation denotes that the current command and $N - 1$ previous commands are merged together to create a new bigger command that hold the historical context [7]. The third improvement is to use more robust representation models like TF-IDF or Doc2Vec model. These proposed representation models will be detailed in this section.[3]

[3]In this chapter, we should distinguish two notations: n-gram is a combination of n **consecutive/adjacency** tokens and N-command is a combination of a command and its $N - 1$ **previous** commands.

N-command

In the BoW model, one command in the dataset is considered as a document. However it can be more useful when we consider a command in its surrounding context. The context around a command is limited to the $N-1$ ***previous*** commands since in the real situation, we can only access the historical input commands. In each SSH session, an N-*command* can be created by taking the current command and applying a sliding window of size N to its previous commands. 1-*command* means that only one command is considered at a time. More specifically, the N-command representation for the command $cmd^{(t)}$ is defined as a list of the command itself and its previous $N-1$ commands $cmd^{(t-1)}, \ldots, cmd^{(t-N-1)}$. An example for the construction of 3-command is illustrated in Fig. 4.1. With this approach, a command (a document) $cmd^{(t)}$ is now represented by a new document $[cmd^{(t-N-1)}, \ldots, cmd^{(t-1)}, cmd^{(t)}]$. In the new document, the input order of the commands is preserved. The new document can be now fed directly to the BoW model or other representation models.

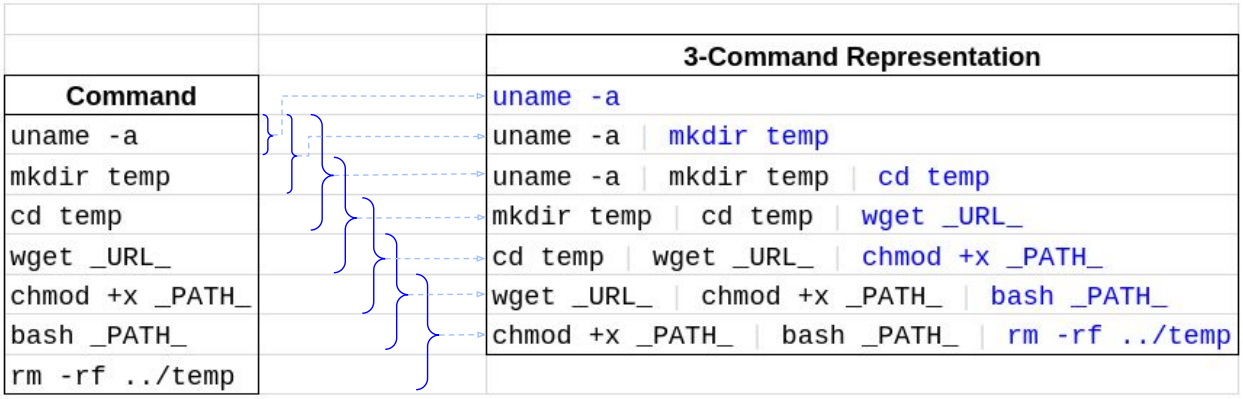


Figure 4.1: Example of N-command representation. From a list of input commands, we group the current command with its two previous commands in their order to create a new set of 3-command documents. In this example, the resource url in the *wget* command is replaced by *_URL_*, the file path used in *chmod* and *bash* commands are replaced by *_PATH_* for readability.

N-command representation can be more robust than 1-command for malicious command detection (as explained in our experiment in the next section), but is it always suitable for different problems? In the binary classification problem, all the commands in

every SSH session are assigned the same label. Taking N commands in the same session together to create a new document is logical and reasonable since this does not change the label of the new document. However, in the multi-class classification problem (where the commands are classified by different risk levels - Chap. 5), the commands in the same session have different labels. Combining a command with its historical commands makes a confused set of labels and it does not help the classification task. For that reason, in our experiments, both 1-command and N-command is used and compared for the binary classification task (Sec. 4.3), and only 1-command is used for multi-class classification task (Sec. 5.4).

TF-IDF Model

In the original BoW model, the input document is represented by a count-vector, in which each element $count(t, d)$ is the raw count of the number of times the word (term) t appears in the given document d. We can see that, even when the stop words and the very common words are removed, there are still several terms that occur much more often than other terms, for example, the keywords or the specific (technical) terms in an article. Moreover, if the document d is long, the term t can have more chances to appear in d. For another document $d^{'}$ having similar content as d but is shorter, the same term t has less chances to appear in $d^{'}$ than in d. This weakness of the count-vector in BoW model comes from the fact that the raw count is not normalized. A simple way to normalize this raw count is to divide it by the count of all other term $t^{'}$ in the document t. The ***term frequency (TF)*** of a term t for the given document d is defined as

$$TF(t, d) = \frac{count(t, d)}{\sum_{t' \in d} count(t', d)}. \tag{4.1}$$

With the normalization defined above, we can only avoid the dominant of the very frequent terms. In one hand, the term frequency of all the term in a document must sum to one. That means if there are many common terms in the document, the values of $f(t, d)$ for the other terms will be low. It is necessary to re-weight $TF(t, d)$. The first weighting function proposed by Karen S. Jones [12] was originally called *term specificity*

and later became the famous ***inverse document frequency (IDF)*** function. The intuition is that "term which occurs in many documents is not a good discriminator, and should be given less weight than one which occurs in few documents" [12].

The idea of the IDF function is derived from Information Theory [33]. The high probability events (the event that occurs very often) have *low information*. For example, the weather has been hot for two weeks in Namur, it is not surprise when we see the weather forecast say it will be 33°C tomorrow. Inversely, the low probability events (the event occurs very rarely) have *high information*. With the same example, if it is said that there will be a thunderstorm and heavy rain tomorrow morning, it will be more surprise. Back to our model, if the terms which appear across many documents in the corpus $\mathcal{C}$ will bring less information since they can not be used to distinguish the different documents. The less-common terms that appear in few documents in the corpus are more informative and can be used to distinguish these documents.

For a discrete event x which has a probability of occurrence $p(x)$, the *Shannon information* of this event is quantified by

$$\text{information}(x) = -\log(p(x)) = \log \frac{1}{p(x)}.$$

The term t plays the role of the event x. The probability $p(x)$ can be interpreted as the probability that the term t appears in any document in the corpus $\mathcal{C}$. And thus the inverse document frequency of the term t in the corpus $\mathcal{C}$ is defined as

$$IDF(t, \mathcal{C}) = \log \frac{|\mathcal{C}|}{1 + |d \in \mathcal{C} : t \in d|}, \tag{4.2}$$

where $|\mathcal{C}|$ is the number of documents in the corpus, $|d \in \mathcal{C} : t \in d|$ is the number of documents in which the term t appears.[4] Finally, the TF-IDF of a term t for a given document d in the corpus $\mathcal{C}$ is defined as

$$\text{TF-IDF}(t, d, \mathcal{C}) = TF(t, d) \cdot IDF(t, \mathcal{C}). \tag{4.3}$$

[4]The denominator is adjusted by adding a constant (1) to avoid the divide-by-zero error when the term t is not indexed in the corpus (and thus it does not appear in any document).

The TF-IDF model represents each document d by a vector of TF-IDF values for every term t in the corpus $\mathcal{C}$. TF-IDF model produces useful representation of documents in such a way that similar documents have similar vector representations (which is often measured by the cosine distance). Thanks to this characteristic, this model is widely used in Information Retrieval [26]. In our experiments, the pipelines using the representation of TF-IDF model give the best accuracy (See more details the next session).

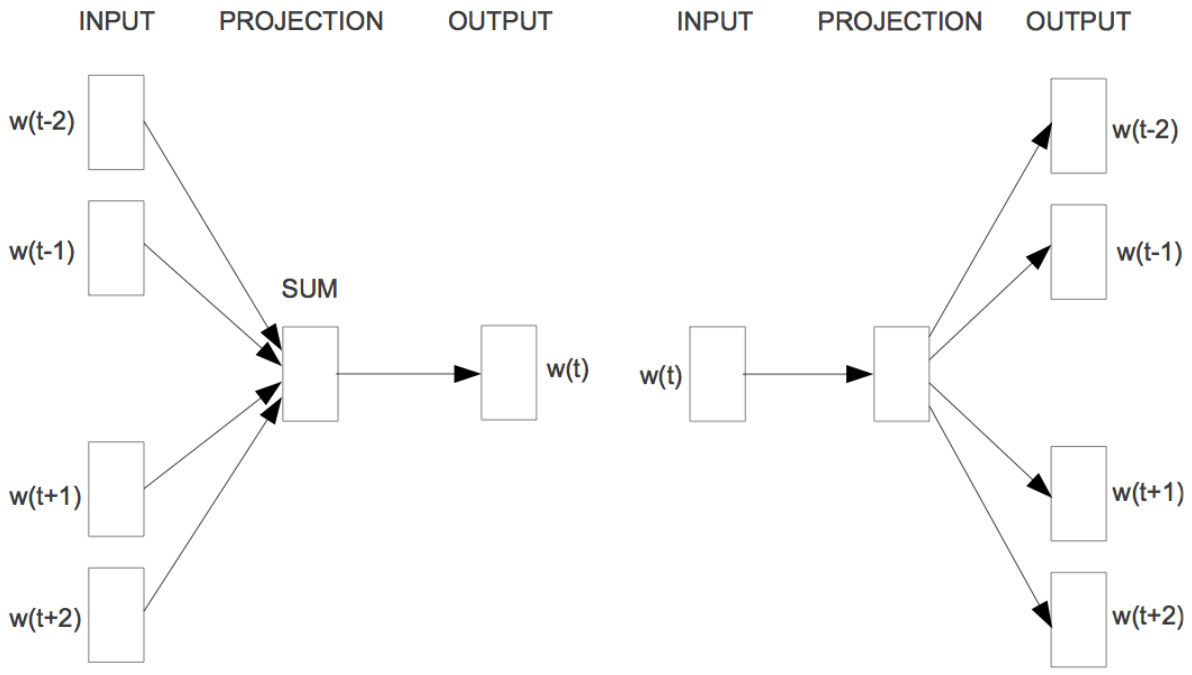
Modern Word2Vec and Doc2Vec Models

With the goal of representing a document by a vector, one naive approach is to represent every word in the document by a vector of same length (e.g., using Word2Vec model [16]) then average these vectors. However, this method does not preserve the order of the words. An other powerful solution is the Paragraph Vector model [13], which is known as Doc2Vec or Sentence2Vec model. It is an unsupervised representation learning method used to transform a text with variate length (a document, a paragraph or a sentence) to a fixed-length feature vector. This model bases on the Word2Vec model, which is presented briefly as follows.

One of the most useful techniques in computational linguistic is to learn a *distributed representation of word in a vector space*, i.e. to represent each word by a vector [1]. *"Ditributed"* here means that, the word is not processed alone but is considered in the surrounding *context*. For example, the word "foot" in the context of the sentence *"Neil Armstrong is ready to plant the first human foot on another world"* [5] means a part of human body. In another context like in the sentence *"Each chair is arranged one foot apart."*, "foot" is a unit of measure. Since the context of a word is important, a word should be represented in a *distributed* manner. Fig. 4.2 illustrates how to represent a word in a distributed manner. To understand this schema, we can think as follows. On one hand, the information of a word $w^{(t)}$ (like its meaning) can be determined by its surrounding words $[w^{(t-2)}, w^{(t-1)}, w^{(t+1)}, w^{(t+2)}]$. This is called Word2Vec *Continuous Bag-of-Word* model in Fig. 4.2a. On the other hand, the information of a word $w^{(t)}$ (like the emotions it brings) can also be distributed (shared)

[5] `https://www.nasa.gov/mission_pages/apollo/apollo11.html`

to its surrounding words $[w^{(t-2)}, w^{(t-1)}, w^{(t+1)}, w^{(t+2)}]$. This is called Word2Vec *Skip-gram* model in Fig. 4.2b.



(a) Word2Vec Continuous Bag-of-Word model. **(b)** Word2Vec Skip-gram model.

Figure 4.2: Illustration of Word2Vec Model.[6]

The Doc2Vec model is exactly the Word2Vec model with an additional document encoding information. In that way, a word will be predicted from not only its surrounding words but also the document containing that word. In our experiment, we use the Doc2Vec implementation in `gensim` [32][7].

4.3 Binary Classification Models

Several reviewed representation models for text are applied to our problem of detection of malicious commands. This section discusses briefly the used classification methods,

[6]The figures are taken from the the blog post of Sebastian Ruder [34] (URL: https://ruder.io/word-embeddings-1)

[7]Gensim Doc2Vec URL: https://radimrehurek.com/gensim/models/doc2vec.html

how to use the feature vector representation of Linux commands with these classification models and how to evaluate the prediction results.

The experiments are performed with two classification methods. The first one is the K-Nearest Neighbor (KNN) model (with K=5). This model is simple, very fast to train since it simply remembers the whole training set, but the prediction time is very slow. The second one is the Support Vector Machine (SVM) [4] model, which is robust and more memory efficient than KNN. SVM model works well with high dimensional data. It will find a hyper-plane to linearly separate the training data. However, the hyper-parameters of this model is more difficult to tune. We use the *LinearSVC* implementation in scikit-learn [24] and find these best hyper-parameters for our dataset: $\{C = 0.1, tol = 1.5e^{-3}\}$[8].

By combining different representations with different classification models, we do the experiments with four different pipelines, which are summarised in Table 4.1.

Context	Feature Representation	Classification	Accuracy	False Negative
1-command	1-gram + BoW	KNN	89.13%	28,878
3-command	1-gram + TF-IDF	LinearSVC	98.13%	2,292
	$\{1, 2, 3\}$-gram + TF-IDF		98.27%	2,066
	$\{1, 2, 3\}$-gram + Doc2Vec		96.71%	**1,449**

Table 4.1: Summary of the experimental results for binary classification problem.

The baseline model (using simple count-vector representation of 1-command BoW model and KNN, similar to one of the models proposed by Dumont et al. [7]) has a low accuracy of 89% on the test set of our own data. It also has a serious problem, that is the *False Negative* error is too high. That means, for this model, there are many (around 28K) malicious commands classified as benign. The detailed classification report can be found in Table 4.2.

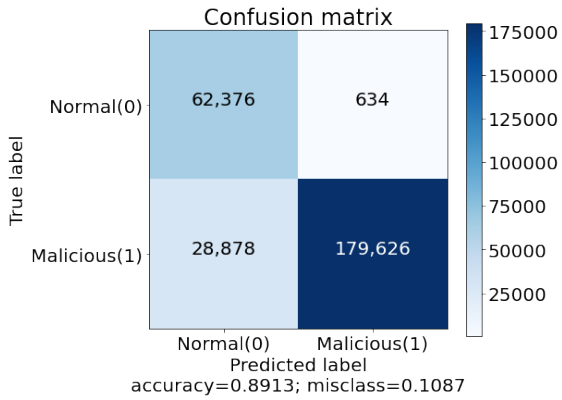
[8] C, one of the most important hyper-parmeters of a linear SVM model, is the coefficient of the regularization ('l2-regularization' is used in our experiment). tol is the tolerance for stopping criteria.

The LinearSVC model is more difficult to train (that requires tuning several hyper-parameters), but the accuracy on the test set is good enough and the prediction is very quick. The quick prediction is helpful when we want to deploy the classification model on real system as we have to make decision in real-time to respond to the input command of the attacker. The three other pipelines in Table 4.1 are trained using LinearSVC with the same hyper-parameters to facilitate the comparison.

As explained in Sec. 3.3, the *False Negative* metric is important for our binary classification problem. This metric measures how many malicious commands are predicted as benign. In real system, if the malicious commands are passed through and executed, they can make harmful impact to the system. We make efforts to reduce this *False Negative* error without reducing the accuracy.

All three proposed pipelines give better results than the baseline model. The first pipeline using 3-command 1-gram with TF-IDF features achieves a good accuracy on the test set (Table 4.3). The *False Negative* is enormously reduced from more than 28K cases to $2,292$ cases. The second pipeline uses $\{1,2,3\}$-gram, slightly reduces the *False Negative* to $2,026$ cases (Table 4.4). Finally, the last pipeline uses Doc2Vec embedding, reduces the *False Negative* to $1,449$ cases (Table 4.5) with a small decrease of performance. In conclusion, the last model is considered as the best one since it achieves a good accuracy while maintaining a very low *False Negative* error.

Baseline model with count-vector



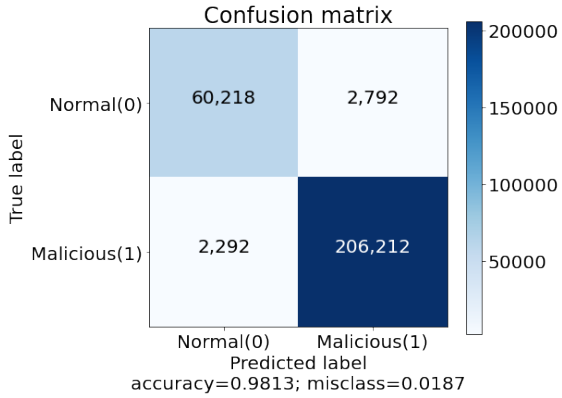
	Precision	Recall	F1-score	Support
Class 0	0.68	0.99	0.81	63010
Class 1	1.0	0.86	0.92	208504
Accuracy			0.89	271514

Highlights

- The model is fast to train but very low in prediction (the disadvantage of KNN model)
- Low accuracy.
- Very high *False Negative*, that is unacceptable in our system.

Table 4.2: Binary classification results with baseline model.

3-command 1-gram, TF-IDF features



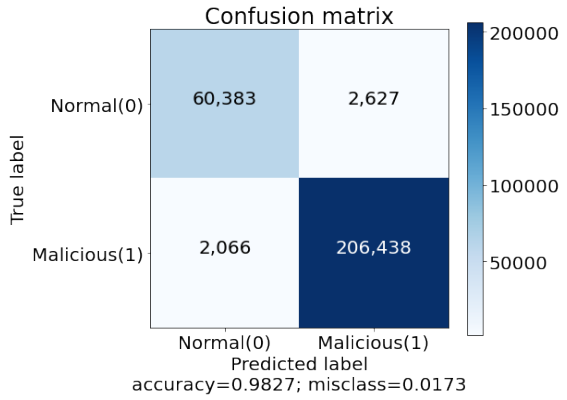
	Precision	Recall	F1-score	Support
Class 0	0.96	0.96	0.96	63010
Class 1	0.99	0.99	0.99	208504
Accuracy			0.98	271514

Highlights

- The accuracy is largely improved in comparison to the baseline model.
- The *False Negative* is also enormously reduced from more than 28K cases to $2,292$ cases.
- The good performance is obtained thanks to the better representation (using 3-command) and the better classification (LinearSVM).

Table 4.3: Binary classification results with 3-command 1-gram and TF-IDF features.

3-command {1,2,3}-gram, TF-IDF features



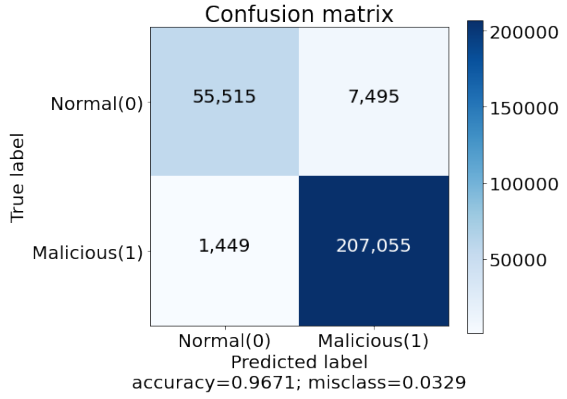
	Precision	Recall	F1-score	Support
Class 0	0.97	0.96	0.96	63010
Class 1	0.99	0.99	0.99	208504
Accuracy			0.98	271514

Highlights

- Using $\{1, 2, 3\}$-gram (a tuple of 3 consecutive tokens) improves slightly the performance.
- The *False Negative* is reduced a little (from $2,292$ to $2,066$ cases).

Table 4.4: Binary classification results with 3-command $\{1, 2, 3\}$-gram and TF-IDF features.

3-command {1,2,3}-gram, Doc2Vec



	Precision	Recall	F1-score	Support
Class 0	0.97	0.88	0.93	63010
Class 1	0.97	0.99	0.98	208504
Accuracy			0.97	271514

Highlights

- The accuracy is slightly lower than the models using TF-IDF features.
- We achieve a lowest *False Negative* of $1,449$ cases among the 4 experimented pipelines.

Table 4.5: Binary classification results with the output of Doc2Vec model.

Summary Chapter *Detection of Malicious Commands in SSH Session*

In this chapter, we formulate the problem as a binary classification problem: identify that a command is benign (class 0 - class negative) or malicious (class 1 - class positive). The bash logs from Github (containing histories of normal users' input commands in their working sessions) are used as examples for the negative class. The commands captured in the Cowrie logs from two honeypots (setup in our project) are used as examples for the positive class. These three important points are discussed:

1. Construction of a robust representation model for text command that transform text into vector of features.
2. Comparison of different pipeline for the binary classification task.
3. Evaluation of the accuracy of the predictive model with a focus on the *False Negative*, an important metric in our malicious command detection problem.

In order to obtain an useful representation of a command, we can consider the context around it. For example, for a current input command $cmd^{(t)}$, we take two previous commands to create a *3-command* representation: $[cmd^{(t-2)}, cmd^{(t-1)}, cmd^{(t)}]$. Each *3-command* is considered as a *document* and we use Doc2Vec model to transform this document into a vector.

There are many binary classification model. The simplest one, a K-Nearest Neighbors model can be trained very quickly but the prediction is very slow. Inversely, the more complicated but powerful model like the linear SVM model is more difficult to train but the prediction is very accurate and efficient.

Our proposed pipeline:

{bash logs + honeypots logs} $\longrightarrow$ Doc2Vec model $\longrightarrow$ Linear SVM classification

gives 97% accuracy and low *False Negative* on the test set.

Chapter 5

Classification of the Commands in SSH Session by Risk Levels

Going beyond the detection of malicious commands, we now tackle a more difficult problem: classifying input commands by risk levels. Simply speaking, we need to estimate how is risky a command to our system in order to help a smart honeypot make decision. For example, if the command has a low risk level, the honeypot can let it pass through and can let it be executed. On the contrary, if the command has a high risk level, the honeypot can block it in order to prevent it from being executed in the real system.

The classification model requires labelled data, which are the pairs of a command and its assigned risk level. In our case, we lack the correct risk level associated with each command, that make the automatic classification task impossible. We have discussed what is the risk level of a command and how to estimate in Sec. 2.2. We present several useful observations in our proper data (in Sec. 5.1) that can help us to find the heuristics to estimate the risk level (in Sec. 5.2). From these heuristic, we can *programmatically* build the training data by constructing the *labeling functions* (details in Sec. 5.3). With the new estimated labels, we build a multi-class classification model and demonstrate its predictions on the real input commands (Sec.5.4). In the following sections, we will assign to each command a level of risk in range from $R0$ to $R4$. $R0$ is the lowest risk level, indicating that a command is not risky. $R4$ is the highest risk level, indicating that a command is

very dangerous. A solution to automatically label the risk level to the dataset (using a novel approach called *Labeling Model*) will also be presented.

5.1 Exploratory data analysis

We have now an unlabelled dataset containing $694,643$ commands captured in the Cowrie logs by two honeypots. We found that there are not much human manual attacks on these honeypots, and the content of the SSH sessions are quite similar.[1] Most of commands are probably sent from automatic programs which scan for the exposed IP address and try to exploit the resource on these machines. Before building a classification model, we must do several *Exploratory Data Analysis* tasks in order to understand our data.

5.1.1 Observation 1: Word Cloud

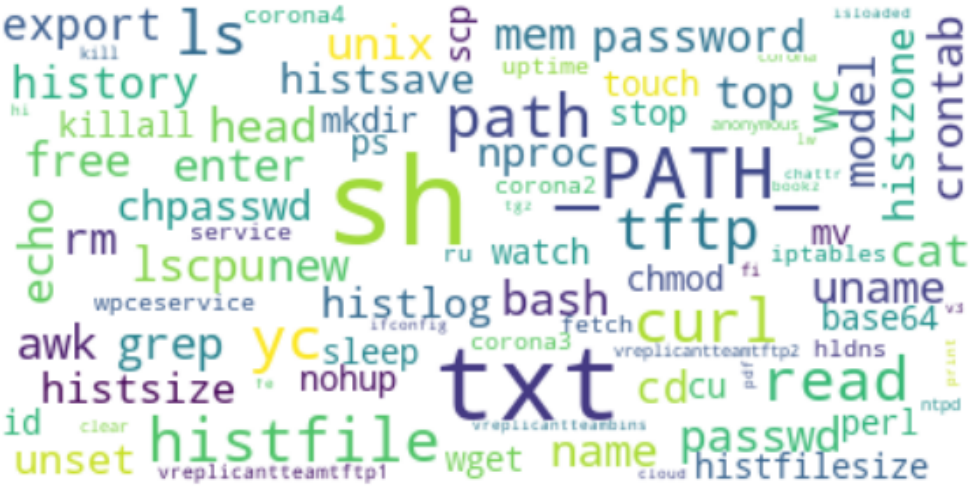


Figure 5.1: Word-cloud of the top 100 frequent tokens parsed from the commands in our dataset. Only the *valid* tokens starting with alphabetic letter are considered.

First of all, from the corpus of input commands captured in the honeypots, we take the top 100 most frequent tokens and visualize them in form of the *word-cloud* as shown in Fig. 5.1. In fact we extract the alphabetic tokens from the text commands, remove the low

[1]Thanks to Mr. Touch for the preliminary analysis and the discussion.

frequent (uncommon) tokens and use the size of the text to present the relative frequency of each token. This word-cloud helps us to see the common commands (or tokens) used in the attack sessions. It is clear that the frequency of the commands does not tell anything about the their risk level. However it gives us the first intuition about what happened in a remote session: maybe some scripts was downloaded with `curl` and executed by `bash/sh`, and all the traces were deleted by touching the history with `histfilesize`. This helps us to draw our attention to the most used commands, without worry about all the available commands in the system. [2] We can therefore invite the experts in Security domain to estimate the risk level only for an important subset of Linux commands for our system.

5.1.2 Observation 2: Topic Modeling

In addition, it can be helpful if we can see different group of input commands in our system. This type of analysis is unsupervised since we can simply regroup the commands by their similarities. For the text data, we apply a topic modeling method to find the hidden topics (semantic clusters) in the data. *Latent dirichlet allocation* (LDA) model [2] is widely used since it does not only reveal the hidden topics but also provides the most representative words in each topic. The result of LDA model for our data with eight topics is shown in Table 5.1. We can change the number of topics to see different patterns. Technically, the found topics are not good enough to represent all the input commands in our dataset because of two reasons: (i) the command (which is considered as a *document*) is short w.r.t. the usual literature document, and (ii) it is often very hard to see the semantic relationship between the tokens (called *words* in LDA) in a command. However, we can find that the obtained groups are coherent enough. For instance, Topic 1 talks about the commands for downloading like `wget` or `curl` and the relevant pattern related to internet resource like `http:_PATH_` or `_IP_`. Topic 6 talks about the `chmod` command

[2] In the `sunrise.info.fundp.ac.be` server of UNamur, we found **3309** available commands when running the command `compgen -bakc| wc -l`. That includes all bash built-ins, reserved keywords, aliases and custom commands found in `$PATH`. In fact the built-ins commands in a Linux system is not too many, maybe around 160 commands - to the best of our knowledge.

Topic 1	Topic 2	Topic 3	Topic 4
| **http:_PATH_** **wget** **curl** -o **_IP_** perl -q - -s	| _PATH_ **rm** **cd** -rf **passwd** **bash** **grep** **cat** _STRING_	_PATH_ ls **>** uname -la & nohup **2>&1** -s -v	echo **sh** **._PATH_** _STRING_ -e & histfilesize=0 export " _PATH_
Topic 5	**Topic 6**	**Topic 7**	**Topic 8**
mv **ssh1.txt** _IP_ perl **p.txt** lscpu -n .ssh_PATH_ **cu.txt** **w.txt**	**chmod** **+x** **777** **u+x** **0777** **0755** 1 { linux2.6 linux2.4	**ps** **killall** -9 perl _STRING_) touch -l https:_PATH_ _STRING_ **[khelper0]**	**history** rm ssr.sh **unset** -rf **histfile** y.txt teamtftp2.sh _IP_ **histlog**

Table 5.1: Eight topics and their representative keywords detected by LDA model.

and its parameters like `+x, 777` to make a file executable. Topic 8 touches the history with some actions to remove history log file or delete an environment variable controlling the history file size. And so on and so forth.

5.2 Proposed Risk Level Estimation

Based on the observations on the clusters of commands and the summary of top frequent commands in our dataset, we propose a simple way to estimate the risk level of the commands as following:

- Consider a small subset of related Linux commands at a time, for example, the group of network related commands or file permission related commands, etc.

- Within each group, we assign relatively **a similar risk level for the similar or closely-related commands**. For example, if we assign a risk level of 2 to `wget` command, we should assign the same risk level to `curl` command since these two command can be used to achieve a similar task and can be an alternative of each other.
- The risk level assigned to each command does not need to be perfect. The most important property of the estimated risk level is its consistency. As explained before, the similar commands should have similar risk level. It is easy to manually check this property, for example, the commands `less`, `more`, `head`, `tail` should have the same risk level no matter what way to estimate them. The estimated risk level should also be logic. For example, in the same group of user-related commands, the command `who` (for displaying who are logged-in) should be less risky than the command `userdel` (for deleting an user in the system).

In that way, we end up with the following proposed groups of closely related basic Linux commands and their estimated risk level: [3]

- Basic commands related to file and directory manipulation and simple text edition (Table 5.2).
- Hardware information related commands (Table 5.3).
- System information related commands (Table 5.4).
- User information related commands (Table 5.5).
- Access control, remote control, file permission and environment variable related commands (Table 5.6).
- Process/Service/Daemon related commands (Table 5.7).
- Network related commands (Table 5.8).

More over, after progressively modifying the above groups multiple times, we found that we can add several simple heuristics to identify a risky command.

[3]The important point to keep in mind is that, the proposed risk level estimation is not perfect, it is subjective, it may have many controversial points. All of these defects come from the limited knowledge of mine own. However it can be modified and improved easily thanks to the programmatic labeling approach (which is presented in the next section).

Command	Short Description	Risk Level
`cd`	Change director	R0
`echo`	Display input text/string	R0
`ls`	list files	R0
`pwd`	Print working directory	R0
`wc`	Show number of words, lines	R0
`awk`	Manipulate text file, e.g. text parsing	R1
`cp`	Copy files/directories	R1
`grep`	Search for specific pattern in a file	R1
`gzip`	Compress `.gz` file	R1
`head`	Show first n lines	R1
`less`	Show content	R1
`ln`	Create symbolic link	R1
`locate`	Find (files and directories) for a given name	R1
`more`	Show content	R1
`mkdir`	Create directory	R1
`mv`	Rename file	R1
`sed`	Edit text file	R1
`scp`	Secure copy	R1
`tail`	Show last n lines	R1
`tar`	Compress/decompress files	R1
`touch`	Create new file	R1
`cat`	Append to file	R2
`make`	Utility to help compile source files	R2
`mount`	Mount a device or remote directory	R2
`vi`	Create/edit text file	R2
`rm`	Remove file	R4
`rm -rf`	Recursively remove a directory (no confirm)	R4

Table 5.2: Basic commands and their estimated risk level.

Command	Short Description	Risk Level
`free`	Display free and used memory	R1
`hdparm`	disk data information	R1
`lsblk`	Show information about block devices	R1
`lshw`	List hardware configuration information	R1
`lspci`	Show PCI devices	R1
`lsusb`	Show USB devices	R1
`dmidecode`	Show hardware information from the BIOS	R2
`dmesg`	Show bootup messages	R2
`df`	Show disk info (free and used space)	R2
`du`	Show disk usage	R2
`fdisk`	Show/modify disk partitions	R3

Table 5.3: Hardware information related commands and their estimated risk level.

Command	Short Description	Risk Level
`date`	Show current date/time	R0
`hostname`	Show system hostname	R0
`uname`	Show system information or kernel info	R0
`uptime`	Display how long the system has been running	R0
`lsof`	List of all open files (resources) on the system	R3
`timedatectl`	Change system clock	R3

Table 5.4: System information related commands and their estimated risk level.

- A command that is too long (more than 80 characters).
- A command using mysterious base64 string.
- A command that modifies history, e.g., remove all history (`history -c`) or disable history(`set +o history` or `unset HISTFILE`).
- A command that modifies system packages (install/uninstall), e.g., `apt-get install` or `make install`.
- A command that controls firewall service or modifies firewall configuration/rules.
- A command that disables/stops a services, e.g., `systemctl stop firewalld` or `systemctl disable firewalld`.

Command	Short Description	Risk Level
`id`	Show details of active users	R1
`last`	Show last system login	R1
`w`	Show result of `who` and `uptime` commands	R1
`who`	Show list of logged-in users	R1
`whoami`	Show who you are logged-in as	R1
`adduser`	Add new user	R3
`groupadd`	Add new group	R3
`chpasswd`	Update passwords (for a list of users in the system)	R4
`passwd`	Set/change password for one user	R4
`userdel`	Delete user	R4
`usermod`	Modify properties of user (e.g. change group)	R4

Table 5.5: User information related commands and their estimated risk level.

Command	Short Description	Risk Level
`chmod`	Change file permission	R2
`chmod +x`	Make file executable	R2
`chmod 777`	Assign full permission to everyone	R2
`chown`	Change file ownership	R2
`ssh`	Connect to remote host	R3
`telnet`	Connect via telnet	R3
`su`	switch user	R4
`sudo`	Use root privileges	R4
`chattr`	Change attribute of a file	R4
`set`	Change system environment variables, e.g. `$PATH`	R4
`unset`	Delete env. variables	R4
`source`	Make the changed env. variables take effect	R4
`EXPORT`	Make env. variables available (e.g. be exported to child-processes)	R4

Table 5.6: Access control, remote access, file permission and environment variables related commands and their estimated risk level.

Command	Short Description	Risk Level
`htop`	Show running processes	R1
`top`	Show running processes	R1
`ps`	Show running processes	R1
`kill`	Kill process by id	R4
`killall`	Kill process (and child processes) by labelled name	R4
`pkill`	Kill process by name	R4
`service`	Control daemon, e.g. `service name [start/stop/restart]`	R4
`systemctl`	Control system process	R4

Table 5.7: Process/Service/Daemon related commands and their estimated risk level.

Command	Short Description	Risk Level
`wget`	Download file	R2
`curl`	Download file	R2
`ifconfig`	Display all network interfaces	R3
`netstat`	Scan active listening ports	R3
`dig`	Show DNS information	R3
`host`	Show IP lookup of a domain (like IP address)	R3
`hostname`	Show local IP address	R3
`tcpdump`	Capture packages on an interface	R4

Table 5.8: Network related commands and their estimated risk level.

- A command containing sensitive keywords like ”hack”, ”hacked”, ”hacking”, ”anonymous”, etc.
- A command to execute a shell, python, perl script (e.g., `bash, sh, python, perl`).

5.3 Construction of a Labeling-Model

The groups of related Linux commands defined above and the ensemble of heuristics give us the essential *keywords* and *rules* to estimate the risk level. The question now is how to transfer this prior knowledge into the label for the real dataset **without manually labeling** each command. The solution is *Snorkel* [28][4], a system for programmatically building and managing training datasets. Instead of hand-labeling the data, we will write the labeling functions to express our constraints. For example, given a rule saying that, a command to download an external script from the internet is assigned a risk level of 2/4, a corresponding labeling function (written in Python using `snorkel` package [31]) is as follows.

Example of Labeling Function

```
from snorkel.labeling import labeling_function

@labeling_function()
def lf_download(cmd):
    black_list = ["wget", "curl"]

    return R2 if any(token in black_list for token in cmd.split())
           else UNKNOWN
```

In this example, the risk level 2/4 is defined by `R2` constant, and the `UNKNOWN` will be assigned to the command if it does not contain any of two keywords `wget, curl`. From the given labeling functions, `snorkel` construct a *Labeling Model* that learns from defined rules and heuristics to automatically assign label (the estimated risk level) to the unlabeled data. In theory, the *Labeling Model* can learn the combination of different labeling functions. It constructs a complicate probabilistic model to predict the label probability of a data point given different labeling functions [29]. In the language modeling tasks like sentiment analysis for example, a sentence can be assigned different emotional nuances and the model will learn the weight of different emotions in order to determine the final score [10]. On the contrary, our task is to assign one risk level for each command. The risk level is not a categorical variable (no natural ordering among the categories), but it is

[4]https://www.snorkel.org/

an ordinal variable instead (the order matters). A custom deterministic *Labeling Model* is created to adapt with our task. It simply assigns the highest risk level among all the risk levels assigned by different labeling function for a given command.
For example the command

```
echo IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gV29ybGQhIg== | base64 --decode | bash
```

will be assigned a risk level 3, which is the highest one assigned by:

- the labeling function related to the basic command group (`echo` - risk level 0),
- the heuristic related to base64 string (risk level 3),
- the heuristic related to executing a command (with `bash` - risk level 3),
- the heuristic related to the long command (risk level 2).

A complete list of 30 labeling functions that encode the proposed estimation for the groups of commands is presented in Table 5.2 to Table 5.8 and the additional heuristics are detailed in the supplementary material at the end of this thesis. It should note that, the labeling functions do not need to be perfectly accurate [30]. They can correlated with other labeling function, or can even overlap or conflict with others. And thus command in the dataset can be labeled by one or more labeling function or even by no labeling function.

The proposed keywords and heuristics are types of *weak supervision* information. When the full labels for the entire data is not provided, we can still use the labeling function to represent the domain knowledge (about the Linux commands) in the form of noisy, programmatic rules and heuristics. This approach is different to the traditional rule-based approach. In the rule-based methods, the set of fixed rules must be defined in advance and we are not allowed to have the conflict rules. Also in these methods, the order of the defined rules matter. When the number of rule increases, we have to make more efforts to manage the rules to satisfy the order constraints and to avoid conflicts. With programmatic approach provided by `snorkel`, the rules can be very simple and easy to combine, that makes the maintenance easier.

These 30 functions are summary in the Fig. 5.2. The coverage metric indicates how a labeling function covers the entries of the dataset. This metric is counted when one data point is labeled by at least one labeling function. The coverage of all labeling function do not sum to one since one data point can be covered by more than one labeling functions. If one data point is labeled by more than one labeling function (maybe with the same or with different labels), it will count for the *overlap* metric. When a data point is labeled differently by two labeling function, it will count

	Labeling Function Name	Estimated Risk Level	Coverage	Overlaps	Conflicts
1	basic_cmds_g1	[0]	33.88%	19.30%	19.30%
2	basic_cmds_g2	[1]	30.14%	28.28%	19.14%
3	basic_cmds_g3	[2]	19.66%	19.15%	19.15%
4	basic_cmds_g4	[4]	3.58%	0.00%	0.00%
5	hardware_info_g1	[1]	10.33%	9.19%	0.06%
6	hardware_info_g2	[2]	0.00%	0.00%	0.00%
7	hardware_info_g3		0.00%	0.00%	0.00%
8	system_info_g1	[0]	15.14%	0.06%	0.06%
9	system_info_g2		0.00%	0.00%	0.00%
10	user_info_g1	[1]	4.57%	0.00%	0.00%
11	user_info_g2		0.00%	0.00%	0.00%
12	user_info_g3	[4]	5.77%	5.77%	5.77%
13	file_permission	[2]	0.35%	0.02%	0.00%
14	remote_access		0.00%	0.00%	0.00%
15	access_control	[4]	0.01%	0.00%	0.00%
16	env_variable	[4]	3.70%	0.93%	0.00%
17	process_g1	[1]	4.62%	0.02%	0.00%
18	process_g2	[4]	0.06%	0.05%	0.04%
19	network_g1	[2]	0.40%	0.14%	0.01%
20	network_g2	[3]	0.01%	0.00%	0.00%
21	network_g3		0.00%	0.00%	0.00%
22	lf_long_cmd	[2]	0.22%	0.22%	0.10%
23	lf_base64	[3]	0.61%	0.61%	0.42%
24	lf_history	[4]	1.86%	0.93%	0.00%
25	lf_install	[3]	0.00%	0.00%	0.00%
26	lf_schedue	[3]	4.55%	0.00%	0.00%
27	lf_firewall	[4]	0.02%	0.02%	0.00%
28	lf_disable_services	[4]	0.03%	0.02%	0.00%
29	lf_sensitive_keywords	[4]	0.00%	0.00%	0.00%
30	lf_execute	[3]	5.83%	5.23%	5.04%

Figure 5.2: Summary of the defined labeling functions. These labeling functions are shown by their name, following by three metrics. The most important metric is the coverage, that measures how do the labeling function cover the unlabeled data.

for the *conflict* metric. This is a big advantage of `snorkel` that allows the user to freely define the flexible labeling functions without worrying about the hard constraints of rule conflicts.

Fig. 5.2 shows that, most of the commands in our dataset come from the group of basic command. However, it should consider all other labeling functions of other groups to make sure that we can cover as much as possible the commands in our dataset. The labeling process is very quick, it take around 1.5 minutes to label around 700K commands. Therefore, we can progres-

sively change the keywords, the proposed estimation of risk level and improve the labeling functions. Fig. 5.3 shows a histogram of the number of command for each estimated risk level. That gives the first insight of the distribution of the class labels which can be helpful when building a multi-class classification model.

The most important point to keep in mind that, the performance of the downstream classification model does not depend on the labels, but it depends on the representation of the input data and the model itself. In our work, when we have only unlabeled data, the proposed solution of applying *Labeling Model* helps us encode the prior knowledge (from Table 5.2 to Table 5.8) into our dataset to make the classification task possible.

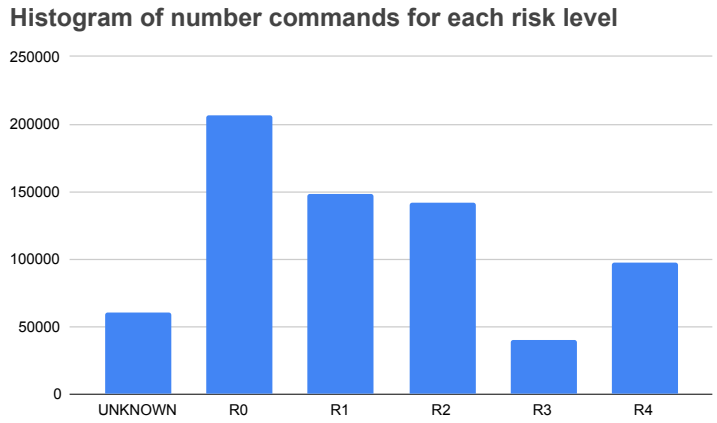


Figure 5.3: Histogram showing the number of command for each estimated risk level.

5.4 Classification of Commands by Estimated Risk Levels

We assume that the estimated risk levels provided by the Labeling Model are reliable enough to use as class labels for our dataset. In order to build a multi-class classification model to classify the command by the estimate risk level, we follow the same pipeline defined in the Chapter 4. From $694,643$ commands in the honeypot logs, 70% of them are used for the training set and 30% of them are reserved for the test set. Firstly, the text data need to be transformed into numerical vectors. We compare two representation models: a simple Bag-Of-Word model and a more powerful Doc2Vec model. These two models are trained only using the corpus in the training set. For the first model, the training and test data are then represented by the Count-Vector. For the second model, the data are represented by the output embedding of the Doc2Vec model. Secondly, we

train a Logistic Regression model[5] with the extracted features produced by these two representation models. We use the Logistic Regression model implement in scikit-learn [24] and tune the C parameter (which control the regularization) and the $solver$ (which determine the strategy of for multi-class classification). The best hyper-parameters found by cross validation and the highlighted classification results are reported in the Table 5.9. The chosen *liblinear* solver [8] makes the Logistic Regression model faster in training phase and also in prediction, that is suitable for real-time prediction in the real system.

	Embedding vector size	Classification model	Accuracy
Model 1	count-vector(128)	*LogisticRegression* $(C = 1e^{-3}, solver = "liblinear")$	95.05%
Model 2	Doc2Vec(128)		99.58%

Table 5.9: Tuned hyper-parameters for representation and classification models. The classification accuracy on the test set can be greatly improved when using the Doc2Vec model.

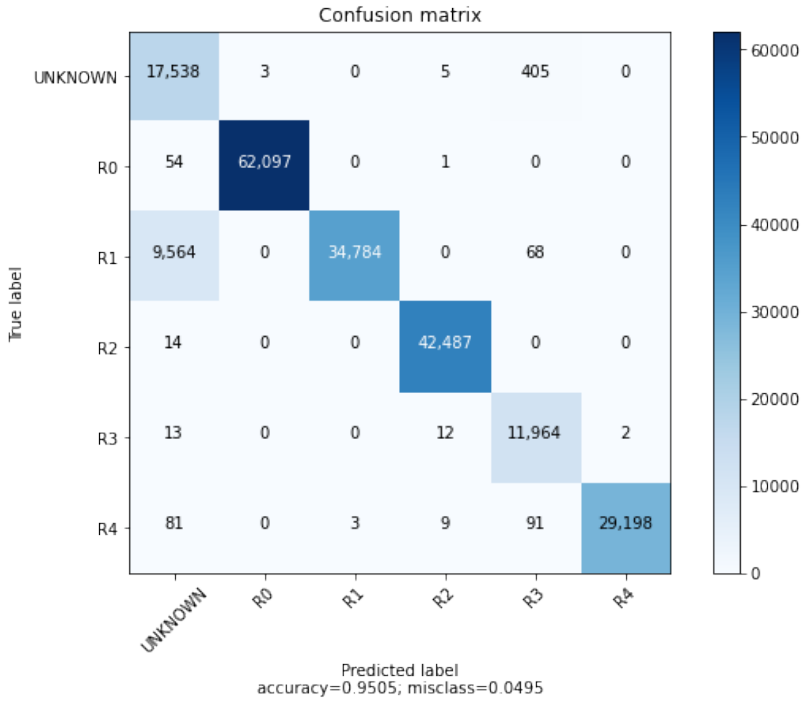


Figure 5.4: Multi-class classification using count-vector as input features.

[5]It should not confuse this model with other one of similar name: Linear Regression is a regression model used to predict real output value. Logistic Regression is a linear (multi-class) classification model.

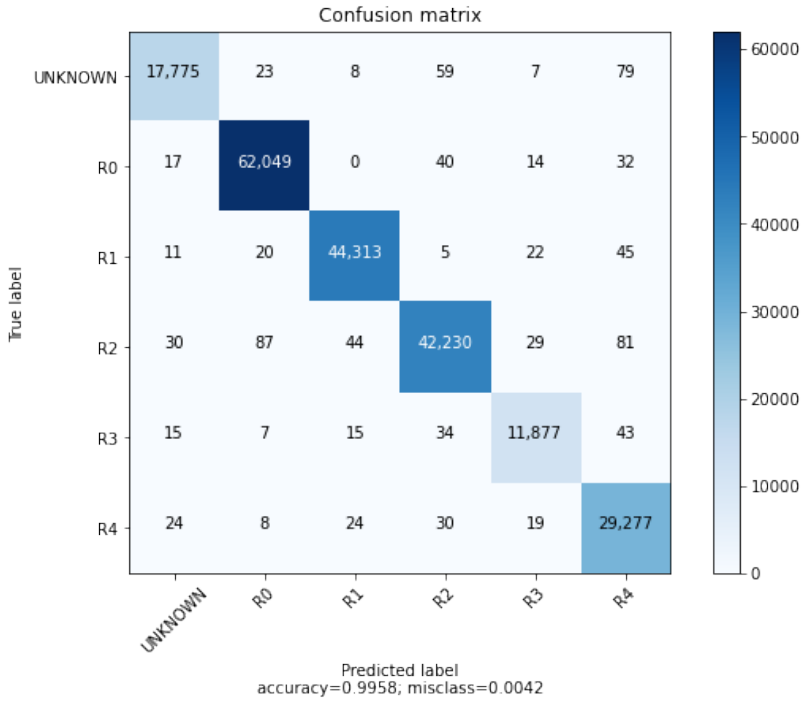


Figure 5.5: Multi-class classification using Doc2Vec features.

In order to evaluate the performance of the classification model, we measure the accuracy of the prediction result and analyze the confusion matrix on the test set. The confusion matrices are detailed in Fig. 5.4 (for the model using Bag-of-Word features) and in Fig. 5.5 (for the model using Doc2Vec embedding). It is clear that, by using a better representation, the accuracy of the classification model can be greatly improved. Moreover, the *False Negative* and *False Positive* error in total are reduced.

The quantitative metrics can be useful for comparing the different models. It can be more helpful to see how the predictive model works with the real input commands. Some examples of input commands and the result if we run these commands in a Linux machine are shown in the Listing *Demo Input Commands*. The following prediction is obtained from the *Model 1* (using simple count-vector features of a Bag-of-Word model + Logistic Regression).[6] The prediction result is shown in Fig. 5.6. It is consistent with the rules defined in Table 5.2, Table 5.8, Table 5.6 and the heuristic of executing a bash script.

[6]The pipeline of Model 1 is faster than Model 2 since the Model 2 takes more time to build the Doc2Vec embedding. In development and evaluation of the proposed risk level estimation, we use the Model 1.

Demo Input Commands

```
$uname -a
Linux xlap2 4.15.0-34-generic #37-Ubuntu SMP Mon Aug 27 15:21:48 UTC 2018
x86_64 x86_64 x86_64 GNU/Linux

$mkdir temp

$cd temp

$wget https://pastebin.com/raw/suLEviDH
--2020-08-14 13:58:27--  https://pastebin.com/raw/suLEviDH
Resolving pastebin.com (pastebin.com)... 104.23.99.190, 104.23.98.190
Connecting to pastebin.com (pastebin.com)|104.23.99.190|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/plain]
Saving to: 'suLEviDH'

suLEviDH        [ <=>                ]    89  --.-KB/s    in 0s

2020-08-14 13:58:27 (3,15 MB/s) - 'suLEviDH' saved [89]

$more suLEviDH
#!/bin/bash

echo IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gV29ybGQhIg== | base64 --decode | bash

$chmod +x suLEviDH

$bash suLEviDH
Hello World!

$rm -rf ../temp
```

Command	Predicted Risk Level
uname -a	R0
mkdir temp	R1
cd temp	R0
wget https://pastebin.com/raw/suLEviDH	R2
chmod +x suLEviDH	R2
bash suLEviDH	R3
rm -rf ../temp	R4

Figure 5.6: Demonstration of model prediction for user input commands.

Summary Chapter *Classification of the Commands in SSH Session by Risk Levels*

In this chapter, we deal with a multi-class classification problem. Each command is now classified by different risk level. The risk level of each Linux command is not well defined since it depends on the usage context. It also requires domain experts' knowledge to be correctly evaluated in different situation.
Classifying a large, unlabeled dataset (of more than 900K commands) is challenging. We propose the following pipeline to solve this problem.

{**UNLABELED** honeypots logs} $\longrightarrow$ Exploratory Data Analysis $\longrightarrow$ Proposed Keywords and Heuristics $\longrightarrow$ Labeling Functions $\longrightarrow$ {**LABELED** text commands} $\longrightarrow$ Doc2Vec $\longrightarrow$ Logistic Regression model.

The most important contribution in this chapter is our labeling functions.

- We start by observing the top frequent commands in the honeypot logs and the different groups of commands provided by a *topic modeling model*.
- We propose coherent and consistent keywords and heuristics that can help us to assign risk level for a given command.
- We then write down an ensemble of **labeling functions** using `snorkel` library to encode our heuristics.
- Thanks to these labeling functions, we construct a noisy-but-consistent **LabelModel** which will assigns an estimated risk level for each command in our large, unlabelled dataset.

This automatic and programmatic way to encode prior knowledge gives us a reliable annotated dataset. We construct a dataset of 6 classes: 5 classes of increasing risk level from R0 to R4 and one class UNKNOWN for the error commands or the commands not covered by our LabelModel. We use Logistic Regression model which outputs 6 scores for each input command, that help determine risk level in an intuitive way. Finally we obtain an accuracy measure of 99.58% on the test set.

Chapter 6

Discussion and Conclusion

In the context of a smart honeypot, we propose a complete workflow to solve the problem of classifying the Linux commands in a SSH session by multiple risk levels. Beside the target of a high accuracy, low *False Negative* and *False Positive* error of the prediction results, we also consider other non-functional requirements in Sec. 6.1. We also discuss briefly what we have done and what we should improve in Sec. 6.2

6.1 Discussion

The first non-functional requirement in our system is the the response time. The predictive model is a part of a decision marker inside the Smart Proxy in a Honeypot. It is required to make prediction with high accuracy and extreme low response time since the honeypot has to return the result in (nearly) real-time. This requirement depends on the classification model. For that reason, the linear SVM model and Logistic Regression model are chosen since they do not have too many hyper-parameters to tune (like a Neural Network trained by Stochastic Gradient Descent) and the training and testing phrases are very fast. [1]

The second non-functional requirement that we care about is the *explainability*. In the multi-class classification task, we apply the *Labeling Model* to automatically assign labels for our dataset. The prediction performance of the system does not depend on how we label the data. However to understand how the *Labeling Model* works and evaluate the quality of the assigned labels, we need

[1]The implementation of these two models in scikit-learn use LIBLINEAR, a fast and high efficient library for large dataset [8]. (https://www.csie.ntu.edu.tw/čjlin/liblinear/, https://github.com/cjlin1/liblinear).

to analyze this model. The interpretability of a complex model is important [17]. We need to know the explanation for the prediction of the model. For example, we can ask: Why does the *Labeling Model* assign such label to a given command? One simple way to do that is to investigate what are the labeling functions that process a given command. Fig. 6.1 shows an example of several commands which are processed by multiple labeling functions. The *Involved Labeling Functions* column shows a list of labeling functions (detailed in the appendix) that process the given command and the corresponding risk level. By observing this type of explanation, we can find out the labeling functions that do not work as expected, update them in order to improve the *Labeling Model*.

Commands	Involved Labeling Functions	
`unset history histfile histsave histzone`	env_variable	[4]
	lf_history	[4]
`service iptables stop`	process_g2	[4]
	lf_firewall	[4]
	lf_disable_services	[4]
`dmidecode | grep vendor | head -n 1`	basic_cmds_g2	[1]
	hardware_info_g2	[2]
`processid=$(ps -u | grep -v _STRING_ | grep cat | grep _STRING_ | head -n 1)`	basic_cmds_g2	[1]
	basic_cmds_g3	[2]
	lf_long_cmd	[2]

Figure 6.1: Analyzing the explainability of the *Labeling Model*: what are the labeling functions that process a given command?

The third non-functional requirement of our system is the ability of maintenance. A good predictive model should generalize well (predict for unseen data). We train the predictive model on a specific set of data collected in the past. These data are processed by two models in our workflow: the *Labeling Model* for automatically assigning labels and the classification model for prediction task. When the new data come and the predictive model does not perform well, we should update the system to learn from these data. One simple strategy is to re-train these two models. In our case, this strategy is feasible since the models are not too large. By design, updating the *La-*

beling Model is easy: we can simply add new labeling functions to express the new rules or new heuristics, re-train and test the new *Labeling Model* to obtain the new set of labels. Since the labeling functions are allowed to be conflicted or overlapped, adding new functions does not affect the old ones. That make the maintenance easy and manageable since we do not have to deal with tedious tasks of managing the dependency, order, priority or conflict among the labeling functions. In contrast, updating the classification model is more difficult. We need some algorithms that can *learn incrementally*: i.e. learn from new training examples (that is called *online learning* or *incremental learning*). The proposed linear classification models (linear SVM and Logistic Regression) can be adapted for online learning by using the optimization procedure based on stochastic gradient descent (SGD). Since the SGD learns from the mini-batches, we can continue to train the old model with the new mini-batches containing the new data. [2]

6.2 Conclusion and Future Work

Applying machine learning to a specific problem is challenging. Our problem is to classify the commands in the SSH session by multiple risk levels. We solve this problem by starting with the simple case of binary classification and then go to the more general case of multi-class classification. Four research questions are proposed to guide our work to focus on four important steps in the workflow: (i) data collection and labeling, (ii) data transformation, (iii) model construction, (iv) evaluation and interpretation of the predicted results. We obtain the good results for both tasks. All of our results are reproducible. [3]

However, there are several points that can be improved in this work. First, we can collect more diverse data which capture more attacking scenarios. Second, the data representation step may also be improved using the modern Recurrent Neural Network (RNN) or the Long Short Term Memory (LSTM) model to learn more powerful representation. Third, we should consider the online learning algorithm in order to update the predictive model with new data. Fourth, we can set weights for the groups of commands presented in Sec. 5.2 according to different requirements. For example, in the web servers the group of network related commands may have higher weights,

[2] scikit-learn support `SGDClassification` for online learning:
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

[3] All of our analyses can be found in this link: `https://www.kaggle.com/thuyngandao/bashlogs`

while in the storage servers the group of commands related to file manipulation may have higher weights.

The last and most important point to improve is in the process of automatic data labeling. From the conceptual aspect, we propose to categorize the Linux commands into groups of related commands. The similar commands in each groups are assigned similar risk levels in order to make sure that the risk levels are relatively coherent. From the practical aspect, the proposed risk levels for each command is encoded into the dataset via the *Labeling Model*. But, we have not yet tackled the problem of evaluating the quality of the automatic labels. Human assessment is needed to verify the predicted labels or to debug the *Labeling Model* (as shown in Fig. 6.1). Another efficient way to do this is to collect a small set of hand-labeled data from experts in the domain, called the *ground-truth*. Data in the ground-truth will be passed through the *Labeling Model*. By evaluating the predicted labels and the ground-truth labels, we can improve *Labeling Model*.

Bibliography

[1] BENGIO, Y., DUCHARME, R., VINCENT, P., AND JAUVIN, C. A neural probabilistic language model. *Journal of machine learning research 3*, Feb (2003), 1137–1155.

[2] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent dirichlet allocation. *Journal of machine Learning research 3*, Jan (2003), 993–1022.

[3] CHESWICK, B. An evening with berferd in which a cracker is lured, endured, and studied. In *Proc. Winter USENIX Conference, San Francisco* (1992), pp. 20–24.

[4] CORTES, C., AND VAPNIK, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[5] DOWLING, S., SCHUKAT, M., AND BARRETT, E. Improving adaptive honeypot functionality with efficient reinforcement learning parameters for automated malware. *Journal of Cyber Security Technology 2*, 2 (2018), 75–91.

[6] DOWLING, S., SCHUKAT, M., AND BARRETT, E. Using reinforcement learning to conceal honeypot functionality. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2018), Springer, pp. 341–355.

[7] DUMONT, P., MEIER, R., GUGELMANN, D., AND LENDERS, V. Detection of malicious remote shell sessions. In *2019 11th International Conference on Cyber Conflict (CyCon)* (2019), vol. 900, IEEE, pp. 1–20.

[8] FAN, R.-E., CHANG, K.-W., HSIEH, C.-J., WANG, X.-R., AND LIN, C.-J. Liblinear: A library for large linear classification. *Journal of machine learning research 9*, Aug (2008), 1871–1874.

[9] FRANCE, A. Ebios risk manager - the method, 2019.

[10] HANCOCK, B., BRINGMANN, M., VARMA, P., LIANG, P., WANG, S., AND RÉ, C. Training classifiers with natural language explanations. In *Proceedings of the conference. Association for Computational Linguistics. Meeting* (2018), vol. 2018, NIH Public Access, p. 1884.

[11] HARRIS, Z. S. Distributional structure. *Word 10*, 2-3 (1954), 146–162.

[12] JONES, K. S. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* (1972).

[13] LE, Q., AND MIKOLOV, T. Distributed representations of sentences and documents. In *International conference on machine learning* (2014), pp. 1188–1196.

[14] LOWE, D. G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision 60*, 2 (2004), 91–110.

[15] LUO, T., XU, Z., JIN, X., JIA, Y., AND OUYANG, X. Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices. *Black Hat* (2017).

[16] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (2013), pp. 3111–3119.

[17] MOLNAR, C. *Interpretable Machine Learning*. 2020.

[18] MURPHY, K. P. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[19] NICOMETTE, V., KAÂNICHE, M., ALATA, E., AND HERRB, M. Set-up and deployment of a high-interaction honeypot: experiment and lessons learned. *Journal in computer virology 7*, 2 (2011), 143–157.

[20] PAUNA, A., AND BICA, I. Rassh-reinforced adaptive ssh honeypot. In *2014 10th International Conference on Communications (COMM)* (2014), IEEE, pp. 1–6.

[21] PAUNA, A., BICA, I., POP, F., AND CASTIGLIONE, A. On the rewards of self-adaptive iot honeypots. *Annals of Telecommunications 74*, 7-8 (2019), 501–515.

[22] PAUNA, A., IACOB, A.-C., AND BICA, I. Qrassh-a self-adaptive ssh honeypot driven by q-learning. In *2018 international conference on communications (COMM)* (2018), IEEE, pp. 441–446.

[23] PAUNA, A., AND PATRICIU, V. V. Casshh–case adaptive ssh honeypot. In *International Conference on Security in Computer Networks and Distributed Systems* (2014), Springer, pp. 322–333.

[24] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[25] POUGET, F., DACIER, M., AND DEBAR, H. Honeypot, honeynet, honeytoken: Terminological issues. *Institut Eurécom (EURECOM), Sophia Antipolis, France, Research Report RR-03-081* (2003).

[26] RAMOS, J., ET AL. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning* (2003), vol. 242, New Jersey, USA, pp. 133–142.

[27] RAMSBROCK, D., BERTHIER, R., AND CUKIER, M. Profiling attacker behavior following ssh compromises. In *37th Annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)* (2007), IEEE, pp. 119–124.

[28] RATNER, A., BACH, S. H., EHRENBERG, H., FRIES, J., WU, S., AND RÉ, C. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases* (2017), vol. 11, NIH Public Access, p. 269.

[29] RATNER, A., HANCOCK, B., DUNNMON, J., SALA, F., PANDEY, S., AND RÉ, C. Training complex models with multi-task weak supervision. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 4763–4771.

[30] RATNER, A. J., DE SA, C. M., WU, S., SELSAM, D., AND RÉ, C. Data programming: Creating large training sets, quickly. In *Advances in Neural Information Processing Systems 29* (2016), pp. 3567–3575.

[31] RÉ, C. Software 2.0 and snorkel: beyond hand-labeled data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018), pp. 2876–2876.

[32] ŘEHŮŘEK, R., AND SOJKA, P. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (Valletta, Malta, May 2010), ELRA, pp. 45–50. `http://is.muni.cz/publication/884893/en`.

[33] ROBERTSON, S. Understanding inverse document frequency: on theoretical arguments for idf. *Journal of documentation* (2004).

[34] RUDER, S. On word embeddings - Part 1. `http://ruder.io/word-embeddings-1/`, 2016.

[35] SHUKLA, M., VERMA, P., AND SCHOLAR, R. Honeypot: Concepts, Types and Working. *Ijedr 3*, 4 (2015), 2321–9939.

[36] SPITZNER, L., AND WESLEY, A. • Examples Honeypots: Tracking Hackers. Tech. rep., 2002.

[37] WAGENER, G., STATE, R., ENGEL, T., AND DULAUNOY, A. Adaptive and self-configurable honeypots. In *12th IFIP/IEEE international symposium on integrated network management (IM 2011) and workshops* (2011), IEEE, pp. 345–352.

[38] ZHANG, Y., JIN, R., AND ZHOU, Z.-H. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics 1*, 1-4 (2010), 43–52.

Supplementary material: Labeling Functions

DAO Thuy Ngan

Definition of different groups of command with estimated risk level.

First, we define the labeling function based on analyzed keywords.

```
keyword_labeling_func_config = {

    # basic command

    "basic_cmds_g1": (R0, ["cd", "pwd", "ls", "wc", "echo", "which"]),

    "basic_cmds_g2": (
        R1, ["grep", "locate", "mkdir", "cp", "scp", "mv", "ln",
             "touch", "more", "less", "head", "tail", "tar", "gzip",
             "awk", "sed"]),

    "basic_cmds_g3": (R2, ["vi", "cat", "make", "mount"]),

    "basic_cmds_g4": ( R4, ["rm", "-rf"]),


    # hardware information related commands

    "hardware_info_g1": (R1, ["free", "lshw", "lsblk", "lspci", "lsusb", "lscpu", "nproc", "hdparm"]),

    "hardware_info_g2": (R2, ["dmesg", "dmidecode", "df", "du"]),

    "hardware_info_g3": (R3, ["fdisk"]),


    # system information related commands

    "system_info_g1": (R0, ["uname", "uptime", "date", "hostname", "cal"]),

    "system_info_g2": (R3, ["lsof", "timedatectl"]),


    # user information related commands

    "user_info_g1": (R1, ["w", "who", "whoami", "id", "last"]),

    "user_info_g2": (R3, ["adduser", "groupadd"]),

    "user_info_g3": (R4, ["usermod", "userdel", "passwd", "chpasswd"]),
```

```
    # access control, remote access, file permission related commands
    "file_permission": (R2, ["chmod", "chown", "+x", "777"]),
    "remote_access": (R3, ["ssh", "telent"]),
    "access_control": (R4, ["sudo", "su", "chattr"]),

    # change system envirionment variables
    "env_variable": (R4, ["set", "unset", "source", "export"]),

    # process related commands
    "process_g1": (R1, ["ps", "top", "htop"]),
    "process_g2": (R4, ["kill", "pkill", "killall", "service", "systemctl"]),

    # network related commands
    "network_g1": (R2, ["wget", "curl", "ip"]),
    "network_g2": (R3, ["ifconfig", "netstat", "dig", "host", "hostname"]),
    "network_g3": (R4, ["tcpdump"]),
}
```

The constants `R0, ..., R4` in the code above denote different risk level and are defined simply as

```
UNKNOWN = -1
R0, R1, R2, R3, R4 = 0, 1, 2, 3, 4
```

Morover, we define the labeling functions based on *heuristics*:

We use several additional heuristics to indicate the malicious commands.

- A command that are too long (e.g., having more than 80 characters).
- A command using mysterious base64 string. (See example above, the base64 string is a bash script that simply echos "Hello World!". When this string is decoded, it can be executed by piping the decoded script to bash.)

```
(17:31):MasterThesis2020$ echo IyEvYmluL2Jhc2gKZWNobyAiICAgIEhlbGxvIFdvcmxkISI= | base64 --decode | bash
 Hello World!
```

- A command that modifies history, e.g., remove all history disable history `unset HISTFILE` (do not remember input commands).
- A command that modifies system packages (install/uninstall), e.g., `apt-get install` or event `make install`.

- A command that controls firewall service or modifies firewall config/rules.
- A command that disables/stops a services, e.g., `systemctl stop firewalld` or `systemctl disable firewalld`
- A command containing sensitive keywords like *"hack", "hacked", "hacking", "anonymous"*, etc.
- A command to execute a script (bash, python, perl script, etc).

```
from snorkel.labeling import labeling_function

@labeling_function()
def lf_long_cmd(cmd):
    # Assign some risk level to a too-long command
    return R2 if len(cmd) > 80 else UNKNOWN


@labeling_function()
def lf_base64(cmd):
    # Using mysterious base64 string
    black_list = ["base64", "--decode"]
    return R3 if any(token in black_list for token in cmd.split()) else UNKNOWN


@labeling_function()
def lf_history(cmd):
    # Maninpulate history
    black_list = ["history", "histfile", "histsize", "histfilesize"]
    return R4 if any(token in black_list for token in cmd.split()) else UNKNOWN


@labeling_function()
def lf_install(cmd):
    # Modify system package (install, uninstall)
    black_list = ["install", "uninstall", "yum", "apt-get", "snap"]
    return R3 if any(token in black_list for token in cmd.split()) else UNKNOWN


@labeling_function()
def lf_schedue(cmd):
    # Automatically schedule jobs
    black_list = ["crontab"]
    return R3 if any(token in black_list for token in cmd.split()) else UNKNOWN


@labeling_function()
def lf_firewall(cmd):
    # Control firewall service or modify firewall config/rules
    black_list = ["firewall", "firewalld", "iptables"]
    return R4 if any(token in black_list for token in cmd.split()) else UNKNOWN



@labeling_function()
def lf_disable_services(cmd):
    # Disable/Stop services, e.g. `systemctl stop firewalld` or `systemctl disable firewalld`
```

```
    black_list = ["disable", "stop"]
    return R4 if any(token in black_list for token in cmd.split()) else UNKNOWN


@labeling_function()
def lf_sensitive_keywords(cmd):
    # Sensitive keywords like "hacked", "anonymous"
    black_list = ["anonymous", "hack", "hacked", "hacking"]
    return R4 if any(token in black_list for token in cmd.split()) else UNKNOWN



@labeling_function()
def lf_execute(cmd):
    # Execute script (e.g., bash or python or perl script)
    black_list = ["./", "._PATH_", "bash", "sh", "perl", "python", "python3"]
    return R3 if any(token in black_list for token in cmd.split()) else UNKNOWN
```

Construction of labeling function using `snorkel`

```
from snorkel.labeling import LabelingFunction


def keyword_lookup(cmd, keywords, label):
    if any(word in cmd.lower() for word in keywords):
        return label
    return UNKNOWN


def make_keyword_lf(f_name, keywords, label=UNKNOWN):
    return LabelingFunction(
        name=f_name,
        f=keyword_lookup,
        resources=dict(keywords=keywords, label=label),
    )
```

Create labeling functions and apply them to the given unlabeld dataset

```
# Create a list of all labeling functions based on keywords

labeling_functions = [
    make_keyword_lf(f_name=f_name, keywords=keywords, label=label)
    for f_name, (label, keywords) in keyword_labeling_func_config.items()
]

# And ones based on heuristics

labeling_functions += [
    lf_long_cmd,
    lf_base64,
```

```
    lf_history,
    lf_install,
    lf_schedue,
    lf_firewall,
    lf_disable_services,
    lf_sensitive_keywords,
    lf_execute

]



# Apply these labeling functions to the unlabeled dataset

from snorkel.labeling import LFApplier

applier = LFApplier(lfs=labeling_functions)
L_train = applier.apply(np.array(cmds_flat))

# ==> `694643it [01:16, 9102.28it/s]`


# Generalte a table to analyze the coverage of each defined labeling function

from snorkel.labeling import LFAnalysis

LFAnalysis(L=L_train, lfs=labeling_functions).lf_summary()
```